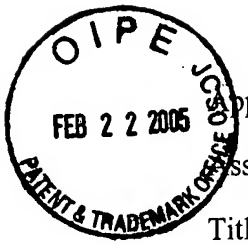


IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant(s):	Siddhartha Nandi, Abhay Kumar Singh, Oleg Kiselev		
Assignee:	VERITAS Operating Corporation		
Title:	System And Method For Dynamically Loadable Storage Device I/O Policy Modules		
Serial No.:	10/717,037	Filing Date:	November 19, 2003
Examiner:	Unassigned	Group Art Unit:	2122
Docket No.:	VRT0091US		

Austin, Texas
February 22, 2005

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

REQUEST FOR RECONSIDERATION OF PETITION TO MAKE SPECIAL UNDER 37 CFR §1.102(d)

Dear Sir:

The applicants hereby request reconsideration of their petition pursuant to 37 CFR §1.102(d) and MPEP § 708.02(VIII) to make the above-identified application special. The original petition was filed on April 20, 2004, and was dismissed November 22, 2004. That dismissal set a shortened statutory period set to expire January 22, 2005. Filed herewith is a Petition for Extension of Time requesting a one-month extension, thereby giving the undersigned a period until February 22, 2005, in which to respond.

The fee of $130.00 for the original petition has already been charged, and no additional fee (beyond the fee for the extension) is believed to be required. Nevertheless, should an additional fee be required, the Commissioner is hereby authorized to charge such a fee to Deposit Account No. 502306.

This renewed petition addresses the Examiner's remarks in his dismissal of November 22, 2004.. Should the Office determine that all the claims presented are not obviously directed to a single invention, the applicants will make an election without traverse as a prerequisite to the grant of special status.

The applicants respectfully submit that a pre-examination search has been performed by a professional search firm in the following classes/subclasses:

Class	Subclasses
G06F	11/20, 11/34, 13/14, 13/16
714	4, 6
H04L	12/56

Enclosed are copies of the following references which are presently believed to be, from among those made of record in previously filed Information Disclosure Statements, the most closely related to the subject matter encompassed by the claims:

US 6,145,028 US 5,944,838 US 2001/0050915
EP 0788055
"Highly Available Storage: Multipathing and the Microsoft MPIO Driver Architecture White Paper," Microsoft Corporation, October 2003

Detailed Discussion of the References

U.S. Patent 6,145,028 (Shank) discloses a hardware independent system and method for adaptively managing and controlling multipath access to an array of data storage devices. Referring to Figure 1, Shank teaches that computer 102 implements a virtual disk driver 134, which translates application 132 I/O requests into I/O virtual disk requests, and also configures the storage device array 104 to implement one or more EMPATH virtual disks 104. The virtual disk driver 134 assigns each of the I/O paths a state of either active (path is available for use in sending I/Os) or passive (an I/O path in the standby mode). At least one of the I/O paths is assigned an active status when the disk is configured. See, e.g., column 4, lines 18-26.

To allow the virtual disk driver 134 to operate with a wide range of storage device array 104 hardware and software implementations, the virtual disk driver 134 comprises a core virtual disk driver 136 and one or more subordinate disk drivers 138. The core virtual disk driver comprises instructions which translate application 132 I/O requests into I/O virtual disk requests. Subordinate disk drivers 138 comprise a series of instructions which implement functionality appropriate for each particular disk array 104 configuration, protocol, and/or hardware implementation. For example, when a storage device array-specific function is required to perform controller switchovers or I/O load balancing, the appropriate subordinate disk driver 138 provides that functionality. In this way, the virtual disk driver 134 provides a generic, hardware-independent solution that supports storage device arrays with open systems. This allows, for example, that the storage device array 104 may be replaced or substituted with one from a different manufacturer, without altering the core virtual disk driver 136. See, e.g., column 4, line 53 through column 5, line 4.

Additionally, Figure 9 illustrates steps performed in one embodiment of Shank's invention incorporating a switch table. First, an inquiry command is sent from the virtual disk driver 134 to the storage array devices 104. This is illustrated in block 902. Next, inquiry data is received 904 from the storage array devices 104. Then, the switch table 140 is scanned 906 to identify a subordinate disk driver 138 corresponding to the storage

array device identification. Finally, the core virtual disk driver 136 is associated 908 with the subordinate disk driver 138. See, e.g., column 6, lines 48-57.

Although Shank teaches the use of a virtual disk driver 134 including a core virtual disk driver 136 and one or more subordinate disk drivers 138, Shank neither teaches nor suggests: (1) receiving a request to load a device policy module into a memory; and (2) loading the device policy module into the memory; and (3) informing the device driver of availability of the device policy module, all as required by claim 1. Similarly, claim 29 recites a computer readable medium including program instructions operable to implement each of the aforementioned operations. As Shank neither teaches nor suggests such operations, the applicants respectfully submit that Shank also does not teach or suggest a computer readable medium including program instructions operable to implement each of the operations.

Regarding claim 14, Shank fails to teach or suggest the claimed storage device discovery module configured to determine information about at least one storage device belonging to a distributed computing system. Moreover, although Shank's virtual disk driver 134 includes a core virtual disk driver 136 and one or more subordinate disk drivers 138, Shank neither teaches nor suggests a driver including an interface configured to communicate with a device policy module that includes at least one of a function, a procedure, and an object-oriented method operable to perform at least one of I/O operation scheduling, path selection, and I/O operation error analysis, also required by claim 14. Similarly, Shank's drivers do not teach or suggest a means for providing storage device specific I/O operation scheduling and communication pathway selection in conjunction with the means for directing I/O operations, as required by claim 37.

Accordingly, the applicants respectfully submit that claims 1, 14, 29, and 37 are allowable over Shank.

U.S. Patent No. 5,944,838 (Jantz) purports to address the problem of providing fast restart of failover of I/O requests from a failed (bad) I/O path to an alternate, operational (good) I/O path, and in a manner that is portable so as to be easily implemented within any host system. See, e.g., column 3, lines 35-40.

In particular, Jantz teaches that a redundant storage control module (also referred to as RDAC or multi-active controller) maintains a queue of pending I/O requests sent for processing via a first asynchronously operating I/O path. In the event of failure of the first asynchronously operating I/O path, the controller restarts the entire queue of pending I/O requests to a second I/O path without waiting for each request to individually fail from the first path. This technique is advantages because some prior techniques required the RDAC module to await failure of each I/O request sent to the failed first I/O path before restarting each failed request on the secondary I/O path, thereby extending the total time required to restart all operations sent to a failed I/O path. Other known techniques provide non-standard features in the lower level driver modules to permit the higher level RDAC modules to directly manipulate dispatch queues maintained for each I/O path within the low level device drivers. See, e.g., the Abstract, column 5, lines 29-63, and column 6, line 57 through column 7, line 24.

Thus, Jantz's RDAC and associated queue neither teach nor suggest: (1) receiving a request to load a device policy module into a memory; (2) loading the device policy module into the memory; and (3) informing the device driver of availability of the device policy module, all as required by claim 1. Similarly, claim 29 recites a computer readable medium including program instructions operable to implement each of the aforementioned operations. As Jantz neither teaches nor suggests such operations, the applicants respectfully submit that Jantz also does not teach or suggest a computer readable medium including program instructions operable to implement each of the operations.

Regarding claim 14, Jantz fails to teach or suggest the claimed storage device discovery module configured to determine information about at least one storage device belonging to a distributed computing system. Neither Jantz's RDAC nor the referenced lower level device drivers teaches or suggest a driver including an interface configured to communicate with a device policy module that includes at least one of a function, a procedure, and an object-oriented method operable to perform at least one of I/O operation scheduling, path selection, and I/O operation error analysis, also required by claim 14. Similarly, Jantz's RDAC and lower level device drivers do not teach or suggest a means for providing storage device specific I/O operation scheduling and

communication pathway selection in conjunction with a means for directing I/O operations, as required by claim 37.

Accordingly, the applicants respectfully submit that claims 1, 14, 29, and 37 are allowable over Jantz.

EP Application 0788055 (Hodges) discloses a data processing system having multiple independent paths for communication between multiple independent storage controllers, and particularly techniques for efficient management of the queues in a multiple independent path storage subsystem where the requests for accessing the storage devices can be carried out without the need for the queues to be in sync with each other.

Referring to Figure 5, Hodges shows array controller 350 in communication with storage devices 345 and storage controller 320. Array controller 350 comprises two identical independently operating controller paths 352 and 354. Controller paths 352 and 354 together provide for enhanced performance and fault tolerant operation. Array controller 350 is in communication with storage controller 320 over a plurality of communication links 380. Array controller 350 includes a plurality of storage interface adapters 400 (four shown in this example, two in each controller path). Storage interface adapters 400 receive commands (instructions, requests) for access to storage devices 345 from storage controller 320 and initially determine if a request issued by storage controller 320 can be satisfied from either buffer 416 or buffer 418. Buffers 416 and 418 store data received from the devices and the storage controller. If the request can be satisfied from the content of either buffer, that information is communicated back to storage controller 320 via one of the storage interface adapters 400. For each request that cannot be satisfied from the content of one of the buffers, one of the storage interface adapters generates a unique request identifier (also referred to as a label) and sends the request with its unique identifier to array managers 412 and 414. Array managers 412 and 414, in turn, add the request and its label to their respective queues 413 and 415. See, e.g., column 8, lines 8-36.

Array managers 412 and 414 are generally responsible for managing data handling between storage controller 320 and storage devices 345. They further translate

data format between storage devices 345 format (native format) and the emulated device format (the format as seen by storage controller 320). Each array manager further maintains its own task queue. For example, array manager 412 maintains array task queue 413 and array manager 414 maintains array task queue 415. Each array manager updates its own array task queue on the basis of messages received from any one of the storage interface adapters 400 and based on the content of the mailbox in each storage device. Array managers 412 and 414 further manage dispatching device operations to storage devices 345 via device interface adapters 440 and 442, respectively. See, e.g., column 8, lines 37-58.

Thus, while Hodges teaches a multipath I/O system using queues that do not need to be synchronized with each other, Hodges neither teaches nor suggests: (1) receiving a request to load a device policy module into a memory; (2) loading the device policy module into the memory; and (3) informing the device driver of availability of the device policy module, all as required by claim 1. Similarly, claim 29 recites a computer readable medium including program instructions operable to implement each of the aforementioned operations. As Hodges neither teaches nor suggests such operations, the applicants respectfully submit that Hodges also does not teach or suggest a computer readable medium including program instructions operable to implement each of the operations.

Regarding claim 14, Hodges fails to teach or suggest the claimed storage device discovery module configured to determine information about at least one storage device belonging to a distributed computing system. Additionally, Hodges' queues (and the manner in which they are serviced) neither teach nor suggest a driver including an interface configured to communicate with a device policy module that includes at least one of a function, a procedure, and an object-oriented method operable to perform at least one of I/O operation scheduling, path selection, and I/O operation error analysis, as required by claim 14. Similarly, Hodges fails to teach or suggest a means for providing storage device specific I/O operation scheduling and communication pathway selection in conjunction with a means for directing I/O operations, as required by claim 37.

Accordingly, the applicants respectfully submit that claims 1, 14, 29, and 37 are allowable over Hodges.

U.S. Patent Application 2001/0050915 (O'Hare) purports to address problems in multipath, multihop systems. In some data storage device arrangements, a first data storage device may be connected to a second data storage device and a processor may only be able to send commands to the second data storage device indirectly using the first data storage device. In other words, the processor only has a direct connection to the first data storage device, and an indirect connection to the second data storage device. If the processor wishes to instruct the second data storage device to perform a data operation, the processor may use a remote system call using the first data storage device. The processor may issue a remote procedure call to the first data storage device which instructs the second data storage device to perform a data operation, for example. This situation can be extended to one or more additional levels of indirect data storage devices. Such indirect communication may require instructions to pass through an indirect connection of multiple data storage devices. There is no way for the processor to, for example, instruct a third data storage device using system calls or remote procedure calls due to the additional storage device layering in the data storage device arrangement. Generally, similar problems may exist in data storage device arrangements that also include more than the foregoing three levels of indirect data storage it device access from a processor. See, e.g., paragraphs 0007-0008.

To address this problem, O'Hare discloses a computer system for providing multihop system calls. Data storage devices are interconnected and also connected to one or more hosts. Each data storage device classifies a data operation as a system call, a remote system call, or a multihop system call. Also described is a multipath multihop system call in which one or more communication paths may be selected using predetermined and/or dynamic communication path selection techniques. The number of communication paths determined may be in accordance with parameters that are included in a multipath multihop system call, tunable system parameters, or a combination of the two. See, e.g., the Abstract.

While describing techniques for implementing multihop communication, O'Hare neither teaches nor suggests: (1) receiving a request to load a device policy module into a memory; (2) loading the device policy module into the memory; and (3) informing the device driver of availability of the device policy module, all as required by claim 1. Similarly, claim 29 recites a computer readable medium including program instructions operable to implement each of the aforementioned operations. As O'Hare neither teaches nor suggests such operations, the applicants respectfully submit that O'Hare also does not teach or suggest a computer readable medium including program instructions operable to implement each of the operations.

Regarding claim 14, while O'Hare describes dynamically determining various pathways, O'Hare fails to teach or suggest the claimed storage device discovery module configured to determine information about at least one storage device belonging to a distributed computing system. O'Hare's multipath multihop system neither teaches nor suggests a driver including an interface configured to communicate with a device policy module that includes at least one of a function, a procedure, and an object-oriented method operable to perform at least one of I/O operation scheduling, path selection, and I/O operation error analysis, also required by claim 14. Similarly, O'Hare fails to teach or suggest a means for providing storage device specific I/O operation scheduling and communication pathway selection in conjunction with a means for directing I/O operations, as required by claim 37.

Accordingly, the applicants respectfully submit that claims 1, 14, 29, and 37 are allowable over O'Hare.

"Highly Available Storage: Multipathing and the Microsoft MPIO Driver Architecture White Paper", Microsoft Corporation, October 2003 (Microsoft) describes the Microsoft MPIO (multipath input/output) solution in Windows 2000 Server, Windows Server 2003, and Windows Storage Server 2003. MPIO is designed to work in conjunction with device specific modules (DSMs) written by vendors. Microsoft provides a sample device-specific module, or DSM, designed to provide a software interface between the multipath driver package and the hardware device. Vendors must

adapt this generic DSM to the specifics of their device or devices. This joint solution allows vendors to design hardware solutions that are tightly integrated with the Windows operating system, and also enables Microsoft to correctly accommodate the non-generic characteristics of each vendor's storage device (such as whether there are multiple active controllers or the controllers have only standby capability), without having to design the MPIO solution in anticipation of each possible difference. See, e.g., page 6, bottom.

The Windows operating system relies on the Plug and Play (PnP) Manager to dynamically detect and configure hardware (such as adapters or disks), including hardware used for high availability/high performance multipathing solutions. The Microsoft MPIO driver development kit is designed to work seamlessly with PnP architecture. The Microsoft MPIO software supports the ability to transparently balance I/O workload, without administrator intervention. MPIO determines which paths to a device are in an active state and can be used for load balancing. Each vendor's load balancing policy (which may use any of several algorithms, such as round robin, the path with the fewest outstanding commands, or a vendor unique algorithm) is set in the DSM. This policy determines how the I/O requests are actually routed. If the DSM returns a path that is inactive, the failover process is initiated. The MPIO driver package, in combination with the vendor DSM, supports end-to-end path failover. The process of detecting failed paths and recovering from the failure, like load balancing, is automatic, extremely fast, and completely transparent to the IT organization. See, e.g., page 8, top, and page 9.

The MPIO driver package consists of three multipath drivers: the port filter driver, the disk-driver replacement, and the bus driver; all of which are implemented in the kernel mode of the operating system. The MPIO driver package works in combination with both the PnP Manager, the disk class driver, the port driver, the miniport driver, and each vendor's device-specific module (DSM) to provide full multipathing functionality. See, e.g., page 11 ("MPIO Drivers").

Although Microsoft teaches a variety of multipath driver features, Microsoft neither teaches nor suggests: (1) receiving a request to load a device policy module into a memory; (2) loading the device policy module into the memory; and (3) informing the

device driver of availability of the device policy module, all as required by claim 1. Accordingly, the applicants respectfully submit that claim 1 is allowable over Microsoft. Similarly, claim 29 recites a computer readable medium including program instructions operable to implement each of the aforementioned operations. As Microsoft neither teaches nor suggests such operations, the applicants respectfully submit that Microsoft also does not teach or suggest a computer readable medium including program instructions operable to implement each of the operations.

Regarding claim 14, although Microsoft does refer to dynamic discovery of devices, it fails to teach or suggest the claimed storage device discovery module configured to determine information about at least one storage device belonging to a distributed computing system. Although Microsoft's MPIO driver packages works in combination with DSMs, Microsoft fails to teach or suggest a driver including an interface configured to communicate with a device policy module that includes at least one of a function, a procedure, and an object-oriented method operable to perform at least one of I/O operation scheduling, path selection, and I/O operation error analysis, as required by claim 14. Similarly, Microsoft fails to teach or suggest a means for providing storage device specific I/O operation scheduling and communication pathway selection in conjunction with a means for directing I/O operations, as required by claim 37.

Accordingly, the applicants respectfully submit that claims 1, 14, 29, and 37 are allowable over Microsoft.

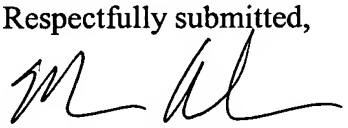
Conclusion

In summary, the applicants respectfully submit that none of the references located during the pre-examination search, or otherwise made of record by the applicants, teach or suggest one or more limitations of each of the applicants' independent claims 1, 14, 29, and 37.

Accordingly, the applicants respectfully request that this petition be granted, and that the present application receive expedited examination. Should any issues remain that might be subject to resolution through a telephonic interview, the Office is requested to telephone the undersigned.

Express Mail Label No:
EV 304740473 US

Respectfully submitted,



Marc R. Ascolese
Attorney for Applicant(s)
Reg. No. 42,268
512-439-5085
512-439-5099 (fax)

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 788 055 B1

(12) **EUROPEAN PATENT SPECIFICATION**

(45) Date of publication and mention of the grant of the patent:
17.09.2003 Bulletin 2003/38

(51) Int Cl.[7]: **G06F 13/16**, G06F 3/06

(21) Application number: **97300431.0**

(22) Date of filing: **23.01.1997**

(54) Multipath i/o storage systems with multipath i/o request mechanisms

Mehrweg-Ein/Ausgabespeichersysteme mit Mehrweg-Ein/Ausgabeanforderungsmechanismus

Systèmes de mémoire d'entrées/sorties à trajets multiples avec mécanismes de demandes d'entrées/sorties à trajets multiples

(84) Designated Contracting States:
DE FR GB

(30) Priority: **05.02.1996 US 597092**

(43) Date of publication of application:
06.08.1997 Bulletin 1997/32

(73) Proprietor: **INTERNATIONAL BUSINESS MACHINES CORPORATION**
Armonk, NY 10504 (US)

(72) Inventors:
- **Hodges, Paul**
  San Jose, California 95120 (US)
- **Hurley, Michael Garwood**
  Cupertino, California 95014 (US)
- **Ouchi, Norman Kenneth**
  San Jose, California 95120 (US)
- **Shih, Mien**
  Saratoga, California 95070 (US)

(74) Representative: **Ling, Christopher John**
IBM United Kingdom Limited,
Intellectual Property Department,
Hursley Park
Winchester, Hampshire SO21 2JN (GB)

(56) References cited:
EP-A- 0 560 343
US-A- 5 201 053
US-A- 5 455 934

EP 0 788 055 B1

Description

Technical Field

[0001] This invention relates to a data processing system having multiple independent paths for communication between multiple independent storage controllers and storage devices. Specifically, this invention relates to a method and means for efficient management of the queues in a multiple independent path storage subsystem where the requests for accessing the storage devices can be carried out without the need for the queues to be in sync with each other.

Background Art

[0002] Data processing systems (systems) with multiple input/output (I/O) storage subsystems generally have multiple independent communication paths between the processor and each storage device in the system. A typical data processing system 100 having such a feature is shown in Fig. 1. Host 110 generally comprises an application program 112, operating system 114, and an I/O supervisor 116 where the I/O supervisor further includes a host task queue 117 for managing the requests issued by the host. Host 110 further comprises a plurality of I/O channels 118 for communication with storage controller 120. Storage controller 120 generally comprises a plurality of I/O ports 122 for communication with host 110, a shared cache 124 for high performance, and a plurality of controller paths 130 for accessing storage devices 140. Storage controller 120 and storage devices 140 are generally referred to as a storage subsystem.

[0003] In general, if an I/O request issued by host 110 cannot be satisfied by information already stored in cache 124, storage controller 120 will access the appropriate storage device via one of the available controller paths 130 to carry out the I/O request. The data processing system of Fig. 1 in general provides high availability due to redundancy of the storage subsystem, multiple I/O channels, multiple controller paths in the storage controller, and multiple communication links between the storage controller and the storage devices.

[0004] In this type of system, a typical queuing of an I/O request issued by host 110 is carried out as follows: I/O request is initiated by application program 112 and passed to I/O supervisor 116. I/O supervisor 116 receives and adds the request to host task queue 117 which is maintained by the I/O supervisor. When one of the communication links 150 becomes available, I/O supervisor 116 initiates an I/O process for the first request in the queue 117 for which the corresponding device is available.

[0005] In this type of system, since host task queue 117 is the only queue of I/O operation available in the system, all operations are initiated at host 110 and only one operation can be active for any device. Furthermore, that operation must be reported as complete before another operation for that device can be initiated by I/O supervisor 116.

[0006] In this type of architecture, failure of one of the channels 118, communication links 150, device communication links 160 or controller paths 130 does not prevent access to storage devices, but a failure that affects host task queue 117 will cause requests in host task queue 117 to fail and is likely to cause one or more programs to abort. Recovery from failure may involve reexecuting the program on the same host or on a different host system.

[0007] However, in a data system where the storage controller has a cache, it is desirable that operations between cache 124 and storage devices 140 be performed concurrently with operations between cache 124 and host (also referred to as central processing unit (CPU)) 110. For example, a request to write data to storage device 142 issued by host 110 can be considered complete by host 110 when the data has been transferred to cache 124. After that, subsequent operations for device 142 can be executed from cache 124 while the updated data is written into storage device 142.

[0008] An example of a data system where the storage controller has a cache is shown in Fig. 2. Fig. 2 represents an IBM System/390 (host 110) in communication with IBM storage controller 3990 (storage controller 120) which controls the operation of IBM magnetic storage devices 3390 (storage devices 140). In this system, there are four communication paths (data paths) between storage controller 120 and storage devices 140. Each data path comprises a controller path 130 and a device communication link 160 and is available for carrying instructions to perform operations on any of the storage devices 140.

[0009] In order to provide high availability, storage controller 120 is generally divided into two storage subcontrollers 132 and 134. Storage sub-controller 132 comprises a controller task queue 126 and a plurality of controller paths 130 (two controller paths shown in Fig. 2). Storage sub-controller 134 comprises a controller task queue 128 and a plurality of controller paths 130 (two controller paths shown in Fig. 2). Task queues 126 and 128 are replicas of each other. Furthermore, the two storage sub-controllers are designed to be at different power boundaries to improve overall system reliability. Therefore, if one of the storage sub-controllers fails, the other storage sub-controller would continue to execute requests from its task queue thus providing continuous performance although at a reduced level.

[0010] Considering that each request for access to a storage device issued by host 110 may contain several sub-commands which any of the sub-commands may be initiated by one sub-controller and completed by another, the management of the task queues in storage controller 120 becomes an extremely serious and critical issue. For example, an I/O request usually contains two sub-commands, 1) a preparatory state command of

"seek/locate" and 2) a time dependent data transfer command known as "read/write". In the case of this type of an I/O request, either one of the storage sub-controllers may be available to carry on any of the sub-commands at any given time and may indeed service the next operation on the queue.

[0011] Therefore, each storage sub-controller has to keep track of the state of the I/O request for each device to ensure that a read/write command is associated with the correct I/O request before it is carried out by either one of the sub-controllers. That is, there must be a mechanism by which acknowledgment of completion of the seek/locate sub-command is received by both sub-controllers. For example, execution of the read/write sub-command by either one of the sub-controllers might be delayed until both sub-controllers have received acknowledgment from the storage device. However, this approach would make access to storage devices extremely slow, and would also compromise the independence of the storage sub-controllers.

[0012] One way to address this problem is to ensure that the storage sub-controllers operate independently to ensure high availability yet communicate very closely with each other in processing device access requests to improve performance. Such an architecture is shown in Fig. 2 where a request for access to a storage device is replicated by each storage sub-controller and sent by each sub-controller to the device. In this architecture, the first storage sub-controller acquiring the device and establishing a communication path transmits the seek/locate command to the device. The communication path is then disconnected from the device and the storage device begins executing the seek/locate sub-command. Once the operation is complete and the device is ready for data transfer, the device raises a flag or interrupt to inform the storage controller. Once, the device raises a flag or interrupt, the first available sub-controller sensing the interrupt from the device acquires the device and completes the data transfer from the device to the shared cache 124 in storage controller 120.

[0013] Therefore, in this architecture, the storage sub-controllers operate independently to provide high availability yet cooperate very closely to ensure high performance. However, since the communication paths through each storage sub-controller are asynchronous, which means requests may be delayed through one communication path compared to another path, this can very well result in a storage sub-controller executing a request or a sub-command which has already been completed by the other storage sub-controller. This could easily lead to wasted operations which lowers overall storage subsystem performance and at the same time could cause data integrity problems.

[0014] A high speed message passing architecture may be utilized in the system of Fig. 2 between the storage sub-controllers to inform a sub-controller of the: operations executed by the other sub-controller. But even sending messages from one sub-controller to another may be delayed leading to duplication of requests and data integrity problems.

[0015] Furthermore, since a plurality of communication paths exist between storage controller 120 and storage devices 140 and any one of the communication paths that are available may service the next I/O request from either one of the task queues 126 and 128, the two copies of the queues must be kept identical at all times to prevent executing the same I/O request twice. This means a sophisticated and complex locking scheme must be used to ensure that a sub-controller ready for work has exclusive access to both copies of the queues and that both copies of the queue are updated before the lock is released.

[0016] Therefore, while the two storage sub-controllers of storage controller 120 are intentionally independent with respect to hardware failures, the communication between the two sub-controllers is very critical to ensure queue integrity. The dependency on complex and elaborate locking schemes and high speed communication between the sub-controllers of the storage controller results in a slowing down of system performance and can lead to performance bottlenecks.

[0017] One way to eliminate such a bottleneck and dependency on complex locking schemes between the queues is to eliminate storage controller 120 and move the necessary function to each storage device. Such an architecture is shown in Fig. 3 in which a plurality of processors 210 and 212 are in communication with multiple storage devices 220 via a small computer system interface (SCSI) bus 218. In this architecture, each storage device 220 contains a device controller 224 and a storage medium 222. Each SCSI device controller 224 further comprises its own device task queue 226.

[0018] Maintaining a queue of work for a storage device at the device level itself is efficient from the queuing standpoint, but such a device requires additional hardware and software, and must have a sophisticated device controller. Although devices having sophisticated device controllers are available (SCSI devices), there is also a desire to have a simple storage device having no device controller that can be directly connected to a storage controller or a RAID controller or a network-attached data server without the problems associated with managing multiple queues at the storage controller.

[0019] An example of a prior art arrangement can be found in EP 0 560 343 disclosing an I/O control system with an I/O processor storing I/O requests in a queue.

[0020] Therefore, in a storage subsystem having a plurality of storage devices in communication with a storage controller via a plurality of independent I/O communication (data) paths, there is a need for a method and means for coordinating I/O requests maintained in multiple task queues and cancelling completed operations among the independent communication paths without the need for keeping the task queues in the storage sub-controller in complete synchronization with each other, without the need to provide continuous sta-

tus of each storage device and request between the storage sub-controllers, and without the need for high speed communications between the sub-controllers in an attempt to keep the queues in sync with each other.

[0021] Also, in a data processing system having a plurality of independent storage controllers in communication with storage devices via a plurality of independent I/O communication (data) paths, there is a need for a method and means for processing I/O requests maintained in multiple task queues in the controllers and cancelling completed operations among the independent communication paths without the need for keeping the task queues in the controller in complete synchronization with each other, without the need to provide continuous status,of each storage device and request between the storage controllers, and without the need for high speed communications between the storage controllers in an attempt to keep the queues in sync with each other.

Disclosure of the Invention

[0022] In accordance with the present invention, in a data processing system having multiple independent I/O paths between a storage controller and storage devices wherein the storage controller comprises a plurality of storage sub-controllers and wherein each storage sub-controller comprises a task queue (queue), a method and means are disclosed wherein the status of each queue and the I/O request (job) completed or in progress are communicated to the other queues via a register that is maintained at each storage:device. This is done by:

(1) assigning a unique identifier to each I/O request before the request is stored in the queues. The unique identifier for each I/O request stored in the queues will ensure that an I/O request is removed from all copies of the queues when that request is completed through any one of the storage sub-controllers;

(2) maintaining a register known as a "mailbox" in each storage device which can be read and/or written to by any sub-controller that establishes: a communication path with that device. The mailbox contains the unique identifier and the status of the last I/O request executed or the I/O request currently being executed by that device; and

(3) allowing any of the storage sub-controllers to read the content of the mailbox and take the appropriate action depending on the content of the mailbox.

[0023] The appropriate action taken by a storage sub-controller reading the mailbox of a device may result in changing the status of the storage device and updating the mailbox content to reflect the change in the status. The sub-controller reading the mailbox may also update its copy of the queue on the basis of what it just read.

[0024] Therefore, based on the disclosed invention, the copies of the queues are not necessarily in sync with each other. Indeed, the queues could be completely out of sync with each other without causing any duplication of work, data integrity problems, or slowing down the processing of the I/O requests.

[0025] For example, if a request to read data from storage device A is issued by the host, the request is assigned a unique label x by the storage controller before the request is stored in the task queues. Once device A becomes available, the request for access to device A is dispatched through all available communication paths. If path 1 is the first to acquire device A, it issues the seek/locate command to prepare the device for data transfer. Device A then stores the unique label x in its mailbox and sets the state of I/O operation to show that device A has been issued the command to prepare itself for data transfer. Path 1 then releases device A and carries out other operations while device A is preparing itself for data transfer. Assuming that path 2 is the first path to sense that device A is ready for data transfer, path 2 reads device A mailbox to find out that the request with unique label x is in progress, associates the data transfer with the request labelled x in its task queue, and then executes the data transfer. Once the data transfer is completed, the state of that specific I/O request at device A is set to complete to reflect that the request associated with the unique label x has been carried out. Assuming that path 1 now attempts to read device A, once it reads device A's mailbox it will find out that the request associated with unique label x has already been completed through another path. So, path 1 deletes the request associated with unique label x from its own task queue and then releases device A.

[0026] Using a unique label for each I/O request stored in the task queues in conjunction with a mailbox at each storage device prevents the same request from being executed more than once by the storage sub-controllers. At the same time, it allows the task queues to be updated asynchronously without affecting the performance of a storage system. Furthermore, using a unique label for each I/O request in conjunction with a mailbox at each storage device eliminates the need for having sophisticated and complex lock step schemes to ensure that the task queue in each storage sub-controller is in sync with the task queues in the other storage sub-controllers. Using the present invention also eliminates the need for providing high speed direct communication between storage sub-controllers to quickly and continuously communicate the queues' status because task queues need not be in sync with each other.

Brief Description of the Drawings

[0027] The invention will now be described, by way of

example only, with reference to the accompanying drawings, in which:

Fig. 1 is a depiction of a prior art data processing system having a host in communication with a storage subsystem;

Fig. 2 is a depiction of another prior art data processing system having a plurality of task queues at the storage controller;

Fig. 3 is a depiction of another prior art data processing system having a task queue in each device;

Fig. 4 is a depiction of the preferred embodiment of the present invention;

Fig. 5 is a detailed schematic diagram of the array controller shown in Fig. 4;

Fig. 6 is a depiction of a mailbox stored in each storage device;

Fig. 7 is a flowchart of I/O requests handling in the preferred embodiment of the present invention;

Fig. 8 is a depiction of an alternative embodiment of the present invention;

Fig. 9 is a depiction of another alternative embodiment of the present invention; and,

Fig. 10 is a depiction of another alternative embodiment of the present invention.

Detailed Description of the Invention

[0028] With respect to Fig. 4, there is shown the preferred embodiment of the present invention. Data processing system 300, comprises a host 310 in communication with storage controller 320, where the storage controller is in communication with storage array 340. The storage array 340 further comprises an array controller 350 which controls access to storage devices 345. In this embodiment, the storage devices that are normally attached to storage controller 320 are replaced with storage array 340. Storage array 340 then can be treated as a logical IBM 3390 storage device. That is, storage array 340 can allow attachment of standard storage devices to storage controller 320 by providing emulation of previously existing storage devices such as the IBM 3390 storage subsystem. Furthermore, array controller 350 in combination with storage devices 345 can provide extremely high data availability via architectures such as reliable arrays of inexpensive disks (RAID).

[0029] Referring back to Fig. 4, host 310 comprises an application program 312, operating system 314, an I/O supervisor 316, and a plurality of I/O channels 318 for communication with storage controller 320. Storage controller 320: generally comprises a plurality of I/O ports 322 for communication with host 310, a shared cache 324 for high performance, and two storage sub-controllers 331 and 332 for accessing storage array 340. Storage sub-controller 331 comprises a controller task queue 326 and controller paths 333 and 334. Storage sub-controller 332 comprises a controller task queue 328 and controller paths 335 and 336. Storage controller 320 communicates with storage array 340 via a plurality of communication lines 380.

[0030] Fig. 5 shows the detailed schematic diagram of array controller 350 in communication with storage devices 345 and storage controller 320. Array controller 350 comprises two identical independently operating controller paths 352 and 354. Controller paths 352 and 354 together provide for enhanced performance and fault tolerant operation. In the preferred embodiment, array controller 350 is in communication with storage controller 320 over a plurality of communication links 380 (four shown in this example). Array controller 350 further comprises a plurality of storage interface adapters 400 (four shown in this example, two in each controller path). Storage interface adapters 400 receive commands (instructions, requests) for access to storage devices 345 from storage controller 320 and initially determine if a request issued by storage controller 320 can be satisfied from either buffer 416 or buffer 418. Buffers 416 and 418 store data received from the devices and the storage controller. If the request can be satisfied from the content of either buffer, that information is communicated back to storage controller 320 via one of the storage interface adapters 400. For each request that cannot be satisfied from the content of one of the buffers, one of the storage interface adapters generates a unique request identifier (also referred to as a label) and sends the request with its unique identifier to array managers 412 and 414. Array managers 412 and 414, in turn, add the request and its label to their respective queues 413 and 415.

[0031] Array managers 412 and 414 are generally responsible for managing data handling between storage controller 320 and storage devices 345. They further translate data format between storage devices 345 format (native format) and the emulated device format (the format as seen by storage controller 320). For example, the data might be stored on storage devices 345 as a fixed block format (native device format). On the other hand, storage controller 320 may be set up to handle the information in a count-key-data format. Therefore, array managers 412 and 414 provide the translation between these two different types of data formats. Each array manager further maintains its own task queue. For example, array manager 412 maintains array task queue 413 and array manager 414 maintains array task queue 415. Each array manager updates its own array

task queue on the basis of messages received from any one of the storage interface adapters 400 and based on the content of the mailbox in each storage device. Array managers 412 and 414 further manage dispatching device operations to storage devices 345 via device interface adapters 440 and 442, respectively.

[0032] In the preferred embodiment, each array controller:path 352 and 354 includes four device interface adapters. Array managers 412 and 414 further notify device interface adapters 440 and 442, respectively, when a request to access a device or data is available for transmission. Array managers 412 and 414 are also in communication with each other via communication bus 410 through which they may notify each-other of the events that may affect the operations of their respective array controller paths 352 and 354. Array managers 412 and 414 may further include RAID exclusive OR (XOR) functions to maintain parity across storage devices 345 to ensure redundant recording of data and fault tolerant operations.

[0033] As mentioned above, device interface adapters 440 and 442 provide access to devices 345 and control storage devices' operations. Device interface adapters also provide operation status to their respective array manager. Fig. 5 further shows a plurality of storage devices 345 (32 devices shown in this example and labelled SD1 through SD32) each device having a mailbox 500. In Fig. 5, storage devices 345 are arranged in four clusters, each cluster including eight devices. Devices 345 can be accessed independently by array controller paths 352 and 354 through their respective device interface adapters 440 and 442. Each device interface adapter can access four storage devices concurrently, one from each cluster.

[0034] with respect to Fig. 6, there is shown a representation of a device mailbox 500 maintained in each of the storage devices 345. In the preferred embodiment, mailbox 500 comprises an eight-byte field for storing the unique request ID 510 (task ID) and the status 520 (task status) of the latest request that has been received by a storage device from array controller 350. The unique request (task) IDs 510 are preferably, but not necessarily, sequential for ease of handling and operation. Mailbox 500 may also include information such as device interface adapter identification from which the request was received. Mailbox 500 may also include information such as device allegiance.

[0035] With reference to Fig. 7, there is shown a flowchart of device access and queues management operation carried out by array managers 412 and 414 in the preferred embodiment of the present invention. The operation will be described with respect to array manager 412 although it is as applicable to array manager 414.

[0036] Array manager 412 initially determines whether any message with respect to queues' status has been received from array manager 414 via communication bus 410 (block 705). Array task queue 413 is updated accordingly if such a message is received (block 706) otherwise array manager 412 determines whether an interrupt from any of the devices 345 has been received (block 710). If a device interrupt from device n is received, device n is selected and its mailbox is read to determine what is the task ID (request ID) and the status of the latest request dispatched to device n (blocks 712 and 715). Assume that the content of mailbox register indicates that the task with unique sequential ID "R" (referred to simply as task "R") is in progress. Array manager 412 then updates array task queue 413 and deletes all the requests having unique sequential ID less than "R" and updates queue 413 to reflect that task "R" is in progress (block 716). Array manager 412 then issues a command to device n to complete task "R" (block 717) and at the same time updates the mailbox in device n to post "task R complete" (block 718). Array manager 412 then deselects device n and deletes task "R" from its array task queue 413 (block 720).

[0037] Referring back to block 715, if the content of mailbox register in device n indicates that the task "R" has been completed, array manager 412 then updates array task queue 413 by deleting task "R" and all the requests having unique sequential ID less than "R" (block 725). Array manager 412 then checks array task queue 413 to determine if there is any other tasks in the queue for device n (block 726); if there is none, the operation returns to block 700; if there is a task "S" for device n, array manager 412 issues commands to initiate task "S" execution (block 727), the mailbox is updated to reflect that task "S" is in progress (block 728), and device n is deselected and the status of task "S" in queue 413 is updated (block 729).

[0038] Returning back to block 710, if array manager 412 does not receive an interrupt from any device, it then checks queue 413 to determine what is the next task that should be carried out and for what device (block 740). Assume that the next task to be carried out by array manager 412 according to its queue 413 is task "T" for device n. Array manager 412 then issues commands to select device n and read its mailbox to determine the ID and the status of the latest task that was issued to device n (block 742). If the task ID in device n mailbox is less than "T" (block 745) and its status is in progress (block 770), .then an error has occurred and the error is communicated back to array manager 412 (block 775). If the task ID in device n mailbox is less than "T" (block 745) and its status is complete, array manager issues commands to initiate task "T", updates the mailbox for device n to show that task "T" is in progress, and then deselects and updates the status of task "T" in array task queue 413 (blocks 771, 772, and 773).

[0039] If the task ID in device n mailbox is "U" which is greater than "T" (block 745) and the status of task "U" is complete, array manager 412 deletes tasks "T" and "U" from queue 413 (block 755) and determines if there is any other task in queue 413 for device n (block 756); if there is none, the operation returns to block 700; if there is a task "V" in queue 413 for device n, array man-

ager 412 issues commands to initiate task "V" execution (block 757), the mailbox is updated to reflect that task "v" is in progress (block 758), and device n is deselected and the status of task "V" in queue 413 is updated (block 759).

[0040] If the task ID in device n mailbox is "U" which is greater than "T" (block 745) and the status of task "u" is "in progress", array manager 412 then updates array task queue 413 and deletes all the tasks (requests) having task IDs less than "U" and updates queue 413 to reflect that task "U" is in progress (block 751). Array manager 412 then issues a command to device n to complete task "U" (block 752) and at the same time updates the mailbox in device n to post "task U complete" (block 753). Array manager 412 then deselects device n and deletes task "U" from its array task queue 413 (block 754).

[0041] Now referring generally to Figs. 4 through 7, the operation of task queues 413 and 415 and the use of the mailbox will be further explained with the following example. Assume task queue 413 contains unique sequentially numbered requests 25 and 26 for storage device 8 (SD8). The request 25 is at the top of the queue (which means it is the next operation to be carried out). Array manager 412 then issues a command through device interface adapter 440 to see whether storage device 8 is available or is busy. If storage device 8 is busy, that information is communicated back to array manager 412 which may then decide to carry out requests for other devices before making another attempt for communication with storage device 8. On the other hand, if storage device 8 is available, array manager 412 selects device 8 and immediately reads mailbox 500 in storage device 8 to determine what is the status of the last request that was carried out by storage device 8. If mailbox 500 shows that the last request that was completed was request number 24, then array manager 412 issues the seek/locate command to storage device 8 for request 25. It then updates the mailbox to show that request 25 is in progress. This is done by setting task status 520 at p (where p stands for "in progress"). Array manager 412 then deselects storage device 8, updates queue 413 to show that request: 25 is in progress. Array manager 412 may also send a message to array manager 414 via communication bus 410 informing array manager 414 that request 25 is in progress. Array manager 412 then may return to an idle state-or establish communication with :the other devices.

[0042] On the other hand, if array manager 412 selects storage device 8 and after reading mailbox 500 finds out that the seek/locate command for request 25 is already in progress, array manager 412 updates queue 413 to show that request 25 is in progress, it then issues commands to device 8 to complete request 25, updates the mailbox to reflect the change in the status, and deselects storage device 8 and deletes request 25 from queue 413.

[0043] Alternatively, if array manager 412 establishes a communication link with device 8 in order to execute request 25 and after reading device 8 mailbox finds out that the seek/locate is already in progress for request 26, array manager 412 will then conclude that request 25 has already been completed by array manager 414. Array manager 412 then deselects storage device 8 and deletes request 25 from queue 413. It also updates queue 413 to show that request 26 is in progress and then returns to an idle state.

[0044] On the other hand, if array manager 412 attempts to carry out request 25 for storage device 8 and finds out, by reading mailbox 500, that request 26 has already been completed, array manager 412 will then conclude that request 25 has already been completed and that both requests 25 and 26 have been serviced by array manager 414. Array manager 412 then deletes both requests 25 and 26 from queue 413 and then searches queue 413 to determine whether there are any other requests in queue 413 for device 8. If such a request is found, then it will attempt to carry it out. If no such request for storage device 8 is found, it deselects storage device 8 and it searches queue 413 for requests for other devices.

[0045] Therefore, based on the detailed description and the examples provided, it can readily be appreciated that servicing a request having a plurality of sub-commands (two, three or more sub-commands) may be initiated by one array manager using the information in its own respective array task queue and the same request may be further carried out or completed by another array manager. It can also readily be appreciated that several requests which are present in both the array task queues may be serviced by only one of the array managers without the other array manager ever having time to participate in such servicing.

[0046] However, using a mailbox register at each storage device provides an easy and efficient way to provide the status of each request to each array manager in an array controller. Thus, queues in each controller path could be completely out of sync with each other without creating any data integrity problems or a potential for duplicating the same request. Through the use of unique request labelling in conjunction with a mailbox at each register, the status of each request (complete, progress, waiting in the queues) can be communicated between the array managers without the need for a separate high speed communication link or without the need for a complex locking scheme to ensure that the array task queues are in *sync* at all times.

[0047] Fig. 8 shows an alternative embodiment of the present invention. Data processing system 800 comprises a host 810 in communication with storage controller 820 where the storage controller is in direct communication with storage devices 840. In this embodiment, unlike the preferred embodiment shown in Figs. 4 and 5, storage controller 820 directly controls access to devices 840. Host 810 comprises application programs 812, operating system 814, I/O supervisor 816

and a plurality of communication channels 818.

[0048] Storage controller 820 comprises a plurality of I/O ports 822 for communication with host 810, a shared cache 824, and two storage sub-controllers 831 and 832. Storage sub-controller 831 comprises a task queue 826 and controller path 829. Storage sub-controller 832 comprises a task queue 828 and controller path 830. Each controller path further comprises a processor and a memory unit. Storage sub-controllers 831 and 832 are in communication with storage devices 840 via communication channels 846 and 847. Each of the storage devices 840 comprises a mailbox which stores the unique ID and the status of the latest request carried out by each of the storage devices. The process of accessing the devices, reading the mailboxes, and managing the task queues by the storage sub-controllers are similar to the one for the preferred embodiment of the present invention. Note that in this embodiment, task queues 826 and 828 could be completely out of sync with each other without slowing down system operation, without causing data integrity problem or without causing duplication of requests.

[0049] Fig. 9 is a depiction of another alternative embodiment of the present invention similar to the embodiment shown in Fig. 8. In this alternative embodiment, data processing system 900 comprises a storage controller 920 where the storage controller comprises four storage sub-controllers 930, 931, 932, and 933. Each storage sub-controller further has its own task queue. Therefore, in this embodiment, there are four task queues in the storage controller that may service requests issued by host 910. Using the disclosed invention, the task queues could be completely out of sync with each other without creating data integrity problems, duplication of requests at the device level or complicated locking schemes for the queues.

[0050] Fig. 10 is a depiction of another alternative embodiment of the present invention where a host may communicate with storage devices via a plurality of storage controllers connected together in a network. In this alternative embodiment, data processing system 1000 comprises a host 1010, a plurality of storage controllers 1020A, 1020B, ..., and 1020N, and storage devices 1040. Each storage controller comprises a cache and may also comprise one or more storage sub-controllers where each storage sub-controller has its own task queue. Therefore, in this embodiment, a host 1010 request is sent to two or more controllers for processing. The unique task ID of the request is preferably, but not necessarily, generated by host 1010 at the same time that the request is generated and sent to the controllers. If the request cannot be processed from the information already stored in and of the caches, it will be stored in the task queues and will be processed by accessing storage devices 1040 in a manner similar to the preferred embodiment of the present invention. Using the disclosed invention, the task queues in different controllers could be completely out of sync with each other without creating data integrity problems, duplication of requests at the device level or complicated locking schemes for the queues.

[0051] A number of embodiments of the present invention have been described. Nevertheless, it will be understood that various modifications may be made without departing from the scope of the invention. For example, the number of task queues may be any number without imposing any limitation on the invention disclosed herein. The mailbox may be expanded to keep additional information such as the history of the past n requests handled by any device and which path initiated or completed the requests. The mailbox may further include information such as aborted requests and whether the device access is limited to a specific path.

Claims

1. A data processing system (300, 800, 1000), comprising:

   a central processing unit (CPU) (310, 810, 1010);

   a controller (320, 820, 1010) in communication with said CPU, the controller having a queue (326, 826), said queue storing requests received from said CPU, **characterised in that** each of said requests having a unique identification label; and, said data processing system (300, 800, 1000) further comprises

   a storage device (340, 840, 1040) for servicing a request stored in any one of said queues, said storage device having a mailbox (500) for storing the status and the unique identification label of the request being serviced.

2. A data processing system (300, 800, 1000) as claimed in claim 1, comprising a plurality of said controllers (320, 820, 1020), each controller having a queue (326, 826).

3. A data processing system (300, 800, 1000) as claimed in claim 1, further comprising:

   a storage subsystem in communication with said CPU, said storage subsystem comprising said controller, said controller having queues concurrently storing requests received from said CPU.

4. A data processing system (300, 800, 1000) as claimed in claim 1 in which said storage controller has a plurality of storage sub-controllers, each of said storage sub-controllers containing a queue, said queues concurrently storing said requests re-

ceived from said CPU.

5. A data processing system (300, 800, 1000) as claimed in any one of claim 1 to claim 4, wherein said request comprises a plurality of sub-commands.

6. A data processing system (300, 800, 1000) as claimed in claim 5, wherein said mailbox (500) further comprises a status field (520) for storing the status of each sub-command being executed.

7. A data processing system (300, 800, 1000) as claimed in claim 6, wherein said controller (320, 820, 1020) comprises means for reading said mailbox (500) and updating the status of the request being serviced in its queue based on the content of said mailbox.

8. A data processing system (300, 800, 1000) as claimed in claim 7, wherein each request stored in said queues comprises a plurality of sub-commands.

9. A data processing system (300, 800, 1000) as claimed in claim 8, wherein said mailbox (500) comprises a status field (520) for storing the status of each sub-command being executed.

10. A method for use in a data processing system (300, 800, 1000), comprising a host (310, 810, 1010) and a storage subsystem (320, 820, 1020) where said storage subsystem includes a storage controller and a storage device (340, 840, 1040) for processing requests received from the host and where said storage controller includes a plurality of queues (326, 826), the method of updating the status of each of said queues, comprising the steps of:

assigning a unique identification label to each request concurrently stored in said queues;

maintaining (118, 728, 753, 758, 772) a mailbox in said storage device for storing the status of each request being serviced;

reading (712, 742) said mailbox to determine the status of the last request being serviced; and,

updating (706, 716, 720, 725, 729, 751, 754, 755, 759, 773) said queues based on the content of the mailbox.

Patentansprüche

1. Datenverarbeitungssystem (300, 800, 1000), das Folgendes umfasst:

eine Zentraleinheit (CPU) (310, 810, 1010);

eine Steuereinheit (320, 820, 1010), die mit der CPU kommuniziert, wobei die Steuereinheit eine Warteschlange (326, 826) hat, wobei die Warteschlange von der CPU empfangene Anforderungen speichert, **dadurch gekennzeichnet, dass** jede der Anforderungen eine eindeutige Kennung hat; und

wobei das Datenverarbeitungssystem (300, 800, 1000) des Weiteren Folgendes umfasst:

eine Speichereinheit (340, 840, 1040), um eine Anforderung zu bedienen, die in einer beliebigen der Warteschlangen gespeichert ist, wobei die Speichereinheit über einen elektronischen Briefkasten (Mailbox) (500) verfügt, um den Status und die eindeutige Kennung der Anforderung, die bedient wird, zu speichern.

2. Datenverarbeitungssystem (300, 800, 1000) nach Anspruch 1, das eine Vielzahl der Steuereinheiten (320, 820, 1020) umfasst, wobei jede Steuereinheit eine Warteschlange (326, 826) hat.

3. Datenverarbeitungssystem (300, 800, 1000) nach Anspruch 1, das des Weiteren Folgendes umfasst:

ein Speicher-Teilsystem, das mit der CPU kommuniziert, wobei das Speicher-Teilsystem die Steuereinheit umfasst, wobei die Steuereinheit Warteschlangen hat, die von der CPU empfangene Anforderungen gleichzeitig speichern.

4. Datenverarbeitungssystem (300, 800, 1000) nach Anspruch 1, bei dem die Speichersteuereinheit über eine Vielzahl von Speicher-Teilsteuereinheiten verfügt, wobei jede der Speicher-Teilsteuereinheiten eine Warteschlange enthält, wobei die Warteschlangen die von der CPU empfangenen Anforderungen gleichzeitig speichern.

5. Datenverarbeitungssystem (300, 800, 1000) nach einem der Ansprüche 1 bis 4, wobei die Anforderung eine Vielzahl von Teilbefehlen umfasst.

6. Datenverarbeitungssystem (300, 800, 1000) nach Anspruch 5, wobei die Mailbox (500) des Weiteren ein Statusfeld (520) umfasst, um den Status eines jeden Teilbefehls, der ausgeführt wird, zu speichern.

7. Datenverarbeitungssystem (300, 800, 1000) nach Anspruch 6, wobei die Steuereinheit (320, 820,

1020) ein Mittel umfasst, um den Inhalt der Mailbox (500) zu lesen und um den Status der Anforderung, die bedient wird, in ihrer Warteschlange auf der Grundlage des Inhalts der Mailbox zu aktualisieren.

8. Datenverarbeitungssystem (300, 800, 1000) nach Anspruch 7, wobei jede in den Warteschlangen gespeicherte Anforderung eine Vielzahl von Teilbefehlen umfasst.

9. Datenverarbeitungssystem (300, 800, 1000) nach Anspruch 8, wobei die Mailbox (500) ein Statusfeld (520) umfasst, um den Status eines jeden Teilbefehls, der ausgeführt wird, zu speichern.

10. Verfahren zur Verwendung in einem Datenverarbeitungssystem (300, 800, 1000), das einen Host (310, 810, 1010) und ein Speicher-Teilsystem (320, 820, 1020) umfasst, wobei das Speicher-Teilsystem eine Speichersteuereinheit und eine Speichereinheit (340, 840, 1040) enthält, um von dem Host empfangene Anforderungen zu verarbeiten, und wobei die Speichersteuereinheit eine Vielzahl von Warteschlangen (326, 826) enthält, wobei das Verfahren zur Aktualisierung des Status einer jeden der Warteschlangen die folgenden Schritte umfasst:

Zuordnen einer eindeutigen Kennung zu jeder Anforderung, die gleichzeitig in den Warteschlangen gespeichert wird;

Verwalten (118, 728, 753, 758, 772) einer Mailbox in der Speichereinheit, um den Status einer jeden Anforderung, die bedient wird, zu speichern;

Lesen (712, 742) des Inhalts der Mailbox, um den Status der letzten Anforderung, die bedient wird, zu ermitteln; und

Aktualisieren (706, 716, 720, 725, 729, 751, 754, 755, 759, 773) der Warteschlangen auf der Grundlage des Inhalts der Mailbox.

Revendications

1. Un système de traitement de données (300, 800, 1000), comprenant :

une unité centrale de traitement (CPU) (310, 810, 1010) ;

un contrôleur (320, 820, 1010) en communication avec ladite CPU, le contrôleur ayant une file d'attente (326, 826), ladite file d'attente stockant des requêtes reçues de ladite CPU, **caractérisé en ce que** chacune desdites requêtes à une étiquette d'identification unique ;

ledit système de traitement de données (300, 800, 1000) comprend en outre

un dispositif de stockage (340, 840, 1040) pour servir une requête stockée dans l'une quelconque desdites files d'attente, ledit dispositif de stockage ayant une boîte à lettres (500) pour stocker l'état et l'étiquette d'identification unique de la requête desservie.

2. Un système de traitement de données (300, 800, 1000) selon la revendication 1, comprenant une pluralité de dits contrôleurs (320, 820, 1020), chaque contrôleur ayant une file d'attente (326, 826).

3. Un système de traitement de données (300, 800, 1000) selon la revendication 1, comprenant en outre :

un sous-système de stockage, mis en communication avec ladite CPU, ledit sous-système de stockage comprenant ledit contrôleur, ledit contrôleur ayant des files d'attente stockant simultanément des requêtes ayant été reçues de ladite CPU.

4. Un système de traitement de données (300, 800, 1000) selon la revendication 1, dans lequel ledit contrôleur de stockage comporte une pluralité de sous-contrôleurs de stockage, chacun desdits sous-contrôleurs de stockage contenant une file d'attente, lesdites files d'attente stockant simultanément lesdites requêtes ayant été reçues de ladite CPU.

5. Un système de traitement de données (300, 800, 1000) selon l'une quelconque des revendication 1 à revendication 4, dans lequel ladite requête comprend une pluralité de sous-instructions.

6. Un système de traitement de données (300, 800, 1000) selon la revendication 5, dans lequel ladite boîte à lettres (500) comprend en outre un champ d'état (520), pour stocker l'état de chaque sous-instruction en cours d'exécution.

7. Un système de traitement de données (300, 800, 1000) selon la revendication 6, dans lequel ledit contrôleur (320, 820, 1020) comprend des moyens pour lire ladite boîte à lettres (500) et mettre à jour l'état de la requête desservie dans sa file d'attente, d'après le contenu de ladite boîte à lettres.

8. Un système de traitement de données (300, 800, 1000) selon la revendication 7, dans lequel chaque requête stockée dans lesdites files d'attente com-

prend une pluralité de sous-instructions.

9. Un système de traitement de données (300, 800, 1000) selon la revendication 8, dans lequel ladite boîte à lettres (500) comprend un champ d'état (520) pour stocker l'état de chaque sous-instruction en exécution.

10. Un procédé d'utilisation, dans un système de traitement de données (300, 800, 1000), comprenant un hôte (310, 810, 1010) et un sous-système de stockage (320, 820, 1020), dans lequel ledit sous-système de stockage comprend un contrôleur de stockage et un dispositif de stockage (340, 840, 1040), pour traiter des requêtes reçues de l'hôte, et dans lequel ledit contrôleur de stockage comprend une pluralité de files d'attente (326, 826), le procédé de mise à jour de l'état de chacune desdites files d'attente comprenant les étapes consistant à :

assigner une étiquette d'identification unique à chaque requête stockée simultanément dans lesdites files d'attente ;

entretenir (118, 728, 753, 758, 772) une boîte à lettres dans ledit dispositif de stockage pour stocker l'état de chaque requête desservie ;

lire (712, 742) ladite boîte à lettres pour déterminer l'état de la dernière requête desservie ; et

mettre à jour (706, 716, 720, 725, 729, 751, 754, 755, 759, 773) lesdites files d'attente d'après le contenu de la boîte à lettres.

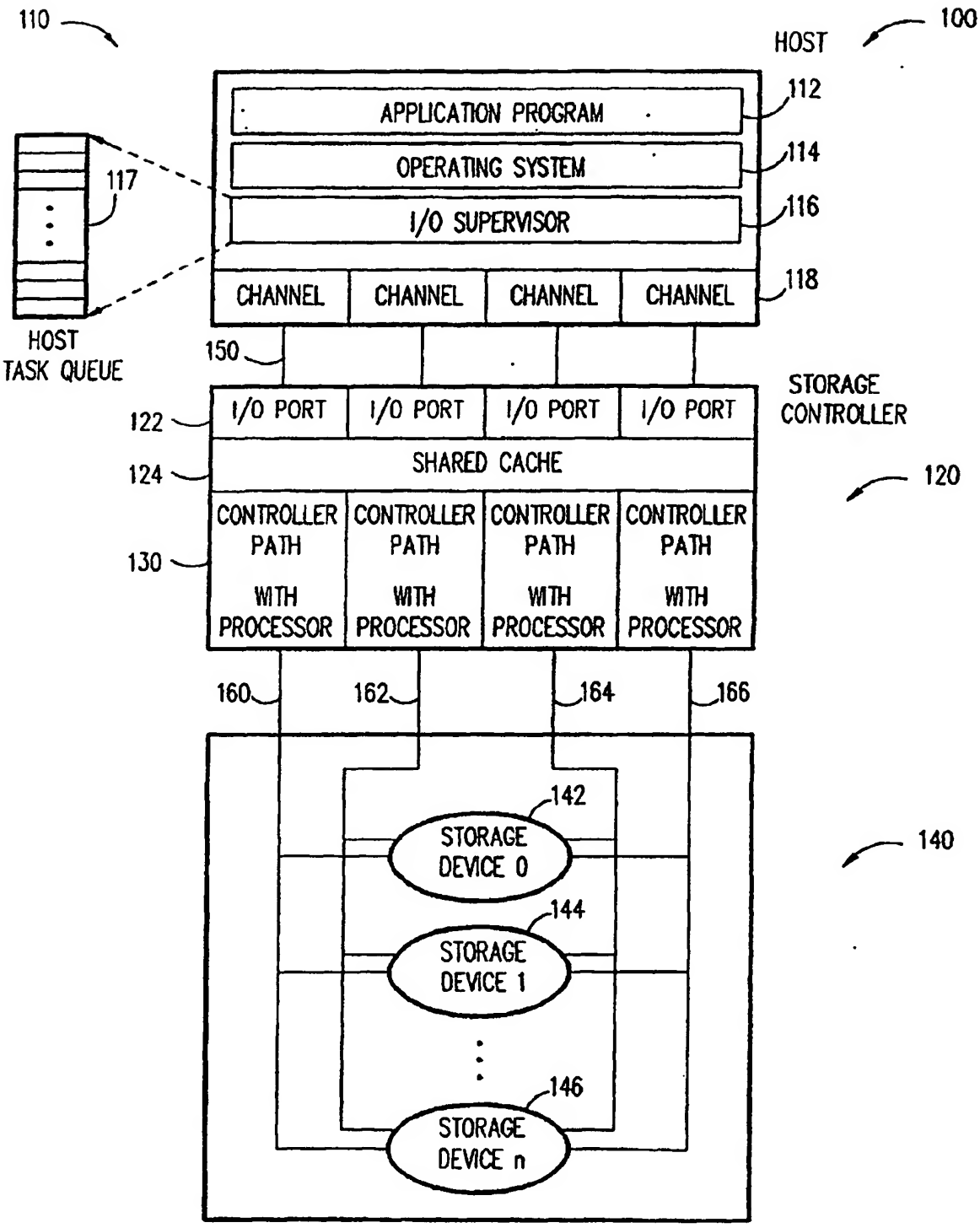

110
HOST
100
APPLICATION PROGRAM
112
OPERATING SYSTEM
114
I/O SUPERVISOR
116
117
CHANNEL
CHANNEL
CHANNEL
CHANNEL
118
HOST
TASK QUEUE
150
STORAGE
CONTROLLER
I/O PORT
I/O PORT
I/O PORT
I/O PORT
122
SHARED CACHE
124
120
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
130
160
162
164
166
142
STORAGE DEVICE 0
140
144
STORAGE DEVICE 1
146
STORAGE DEVICE n

FIG. 1

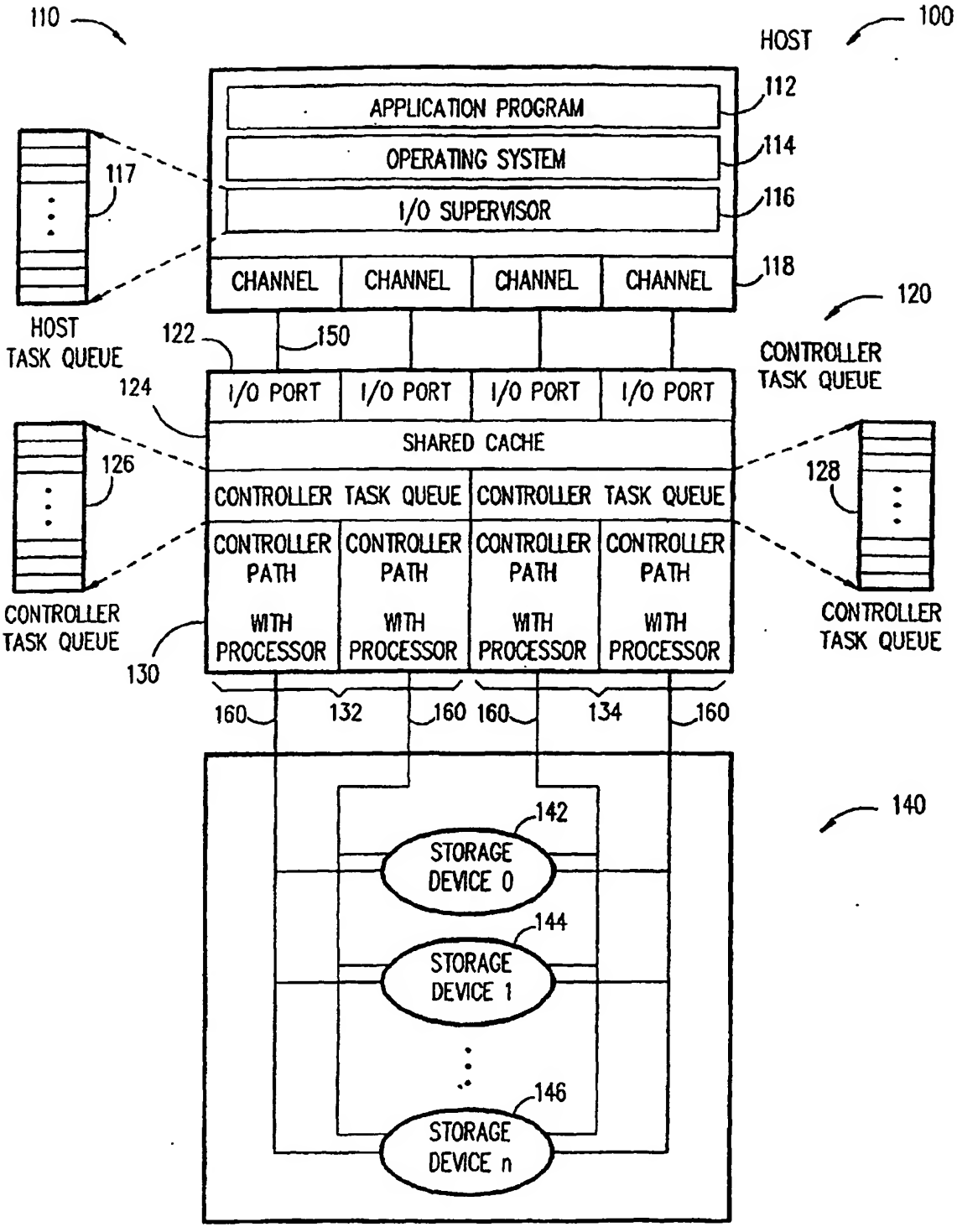

110
HOST
100
APPLICATION PROGRAM
112
OPERATING SYSTEM
114
I/O SUPERVISOR
116
CHANNEL
CHANNEL
CHANNEL
CHANNEL
118
117
HOST
TASK QUEUE
122
150
120
CONTROLLER
TASK QUEUE
124
I/O PORT
I/O PORT
I/O PORT
I/O PORT
SHARED CACHE
126
CONTROLLER TASK QUEUE
CONTROLLER TASK QUEUE
128
CONTROLLER
PATH
WITH
PROCESSOR
CONTROLLER
PATH
WITH
PROCESSOR
CONTROLLER
PATH
WITH
PROCESSOR
CONTROLLER
PATH
WITH
PROCESSOR
CONTROLLER
TASK QUEUE
CONTROLLER
TASK QUEUE
130
160
132
160
160
134
160
140
142
STORAGE
DEVICE 0
144
STORAGE
DEVICE 1
146
STORAGE
DEVICE n

FIG. 2

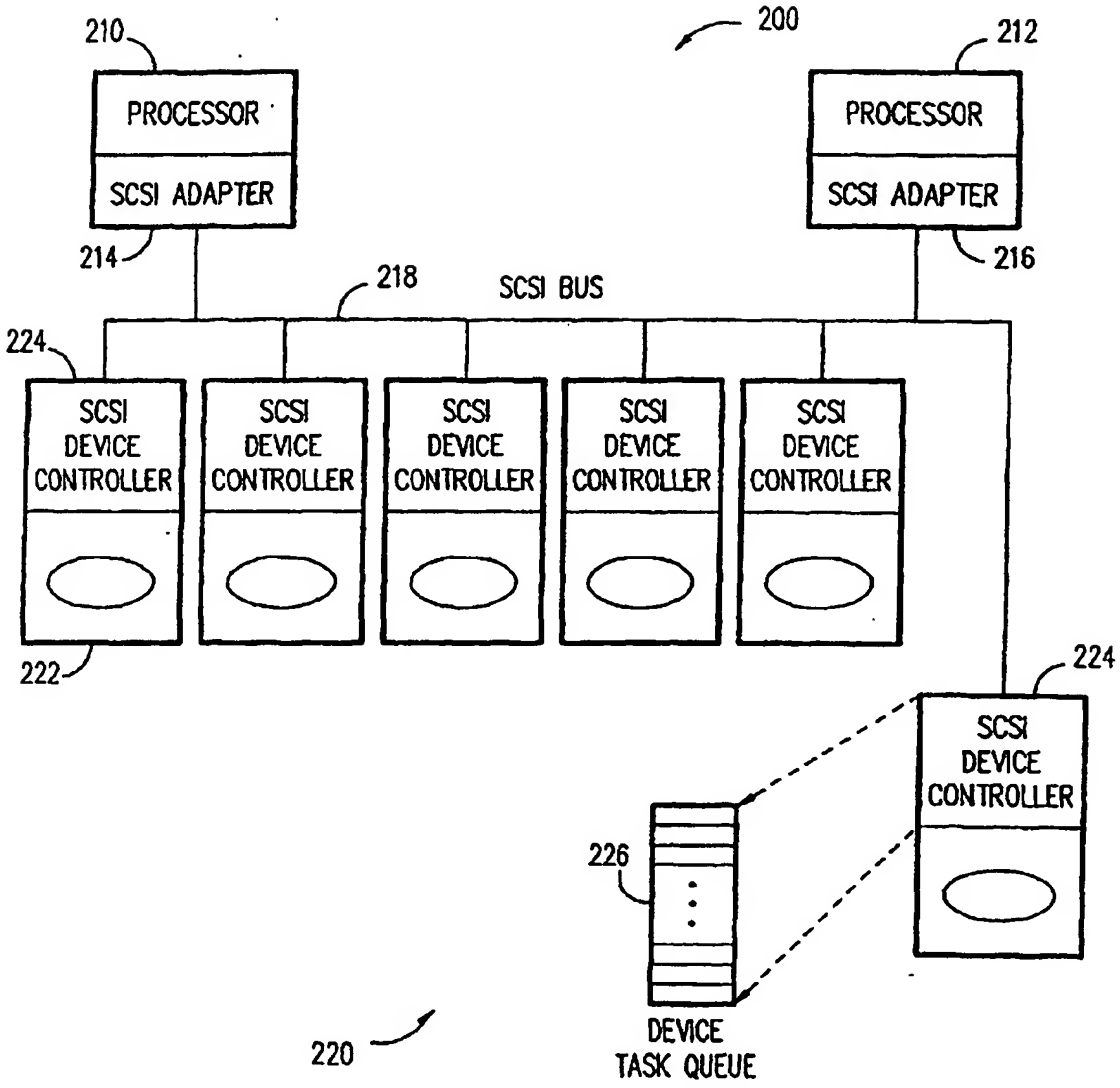

200
210
PROCESSOR
SCSI ADAPTER
214
212
PROCESSOR
SCSI ADAPTER
216
218
SCSI BUS
224
SCSI DEVICE CONTROLLER
SCSI DEVICE CONTROLLER
SCSI DEVICE CONTROLLER
SCSI DEVICE CONTROLLER
SCSI DEVICE CONTROLLER
222
224
SCSI DEVICE CONTROLLER
226
DEVICE TASK QUEUE
220

FIG. 3

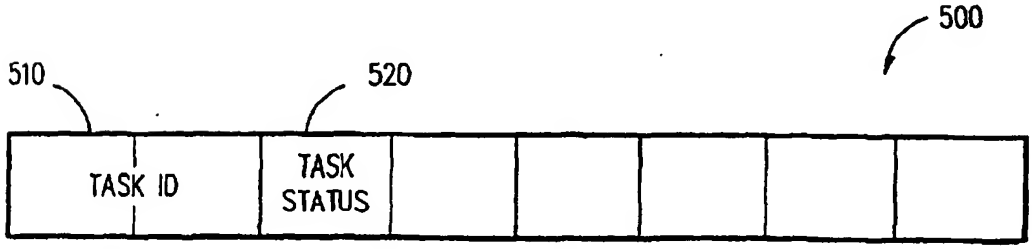

500
510
520
TASK ID
TASK STATUS

FIG. 6

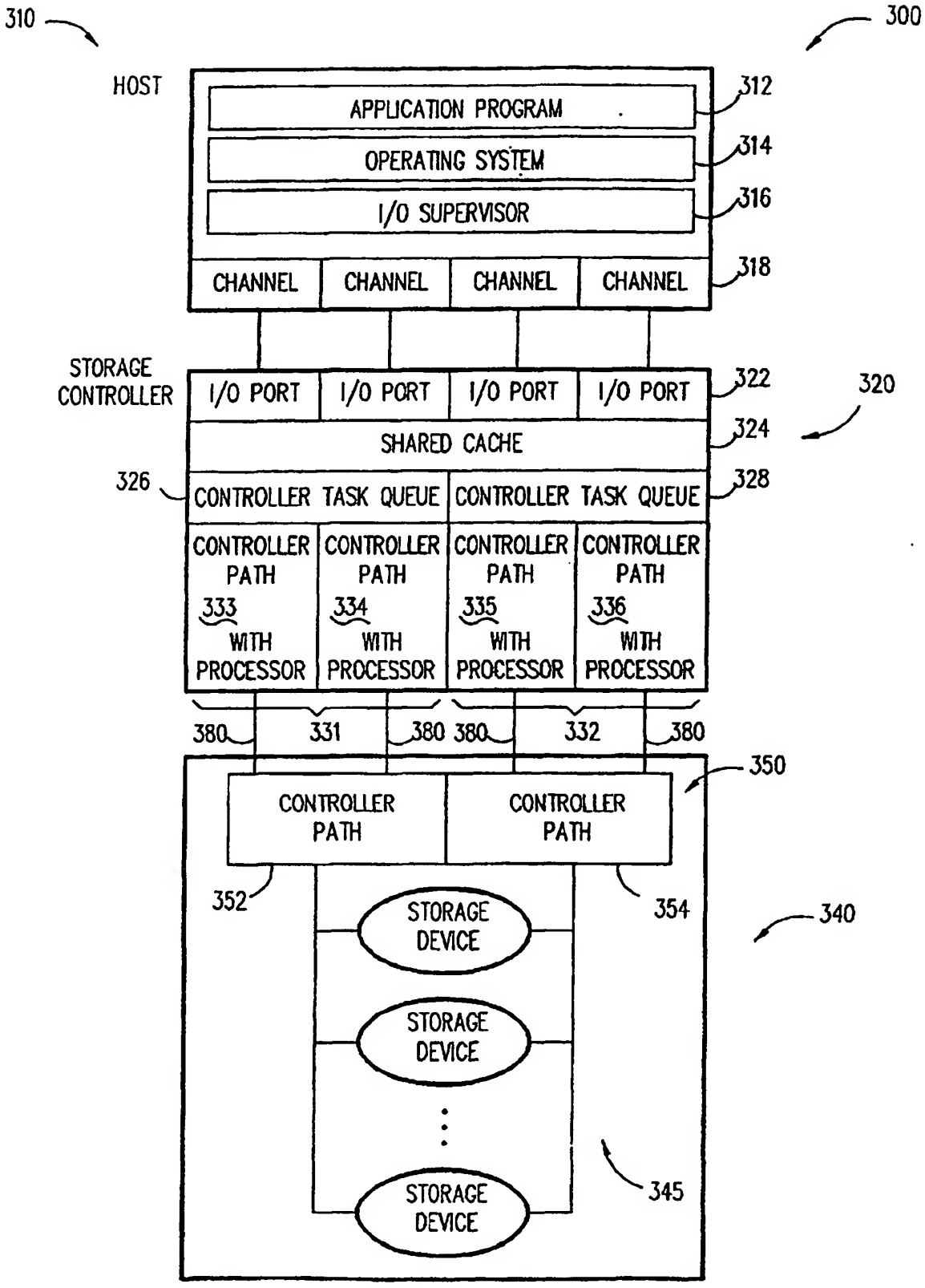

310
300
HOST
APPLICATION PROGRAM
312
OPERATING SYSTEM
314
I/O SUPERVISOR
316
CHANNEL
CHANNEL
CHANNEL
CHANNEL
318
STORAGE CONTROLLER
I/O PORT
I/O PORT
I/O PORT
I/O PORT
322
320
SHARED CACHE
324
326
CONTROLLER TASK QUEUE
CONTROLLER TASK QUEUE
328
CONTROLLER PATH
333
WITH PROCESSOR
CONTROLLER PATH
334
WITH PROCESSOR
CONTROLLER PATH
335
WITH PROCESSOR
CONTROLLER PATH
336
WITH PROCESSOR
380
331
380
380
332
380
350
CONTROLLER PATH
CONTROLLER PATH
352
354
340
STORAGE DEVICE
STORAGE DEVICE
STORAGE DEVICE
345

FIG. 4

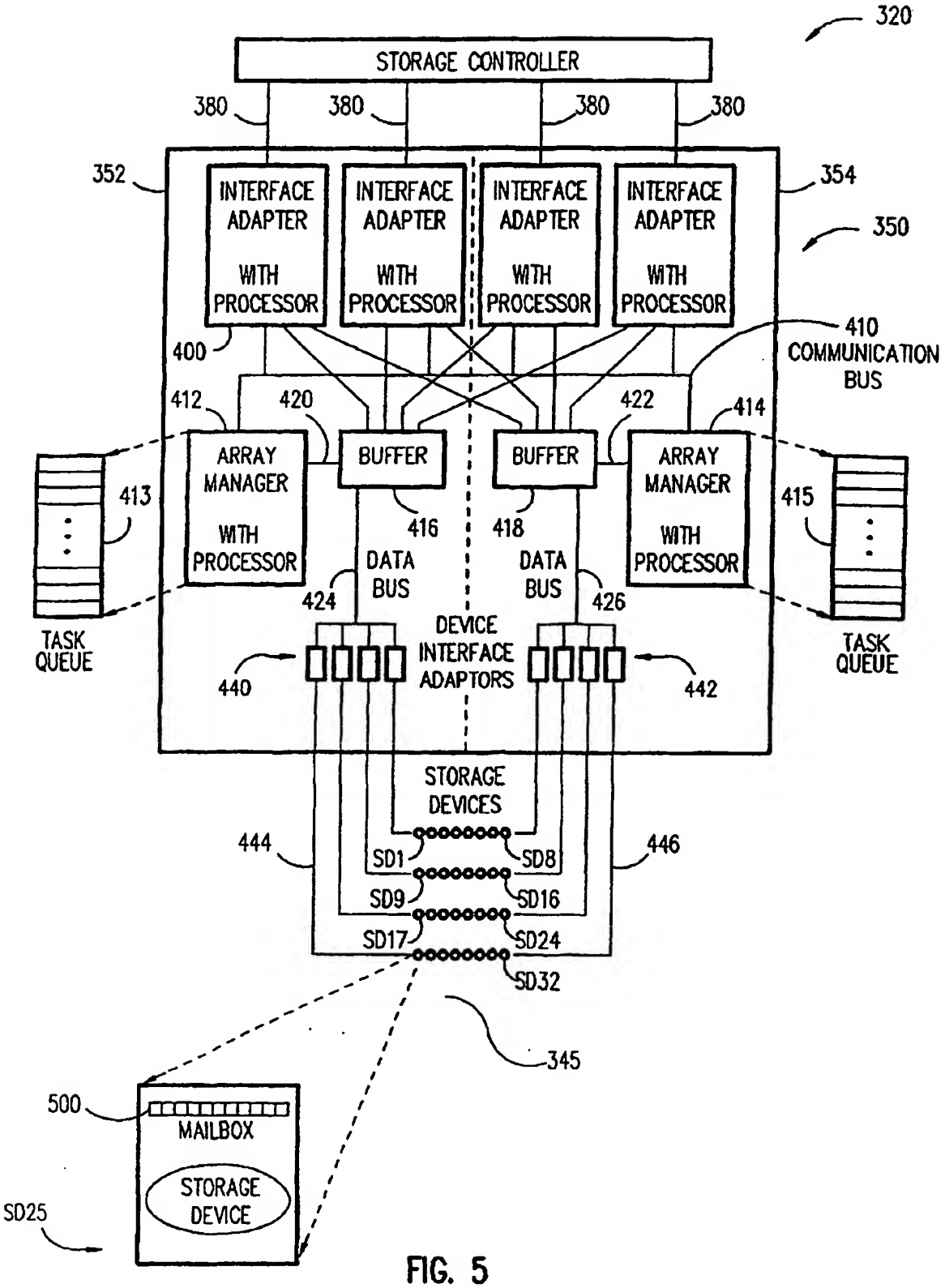

320
STORAGE CONTROLLER
380
380
380
380
352
354
350
INTERFACE ADAPTER WITH PROCESSOR
INTERFACE ADAPTER WITH PROCESSOR
INTERFACE ADAPTER WITH PROCESSOR
INTERFACE ADAPTER WITH PROCESSOR
400
410
COMMUNICATION BUS
412
420
422
414
ARRAY MANAGER WITH PROCESSOR
BUFFER
BUFFER
ARRAY MANAGER WITH PROCESSOR
413
415
416
418
DATA BUS
DATA BUS
424
426
TASK QUEUE
TASK QUEUE
DEVICE INTERFACE ADAPTORS
440
442
STORAGE DEVICES
444
446
SD1
SD8
SD9
SD16
SD17
SD24
SD32
345
500
MAILBOX
STORAGE DEVICE
SD25

FIG. 5

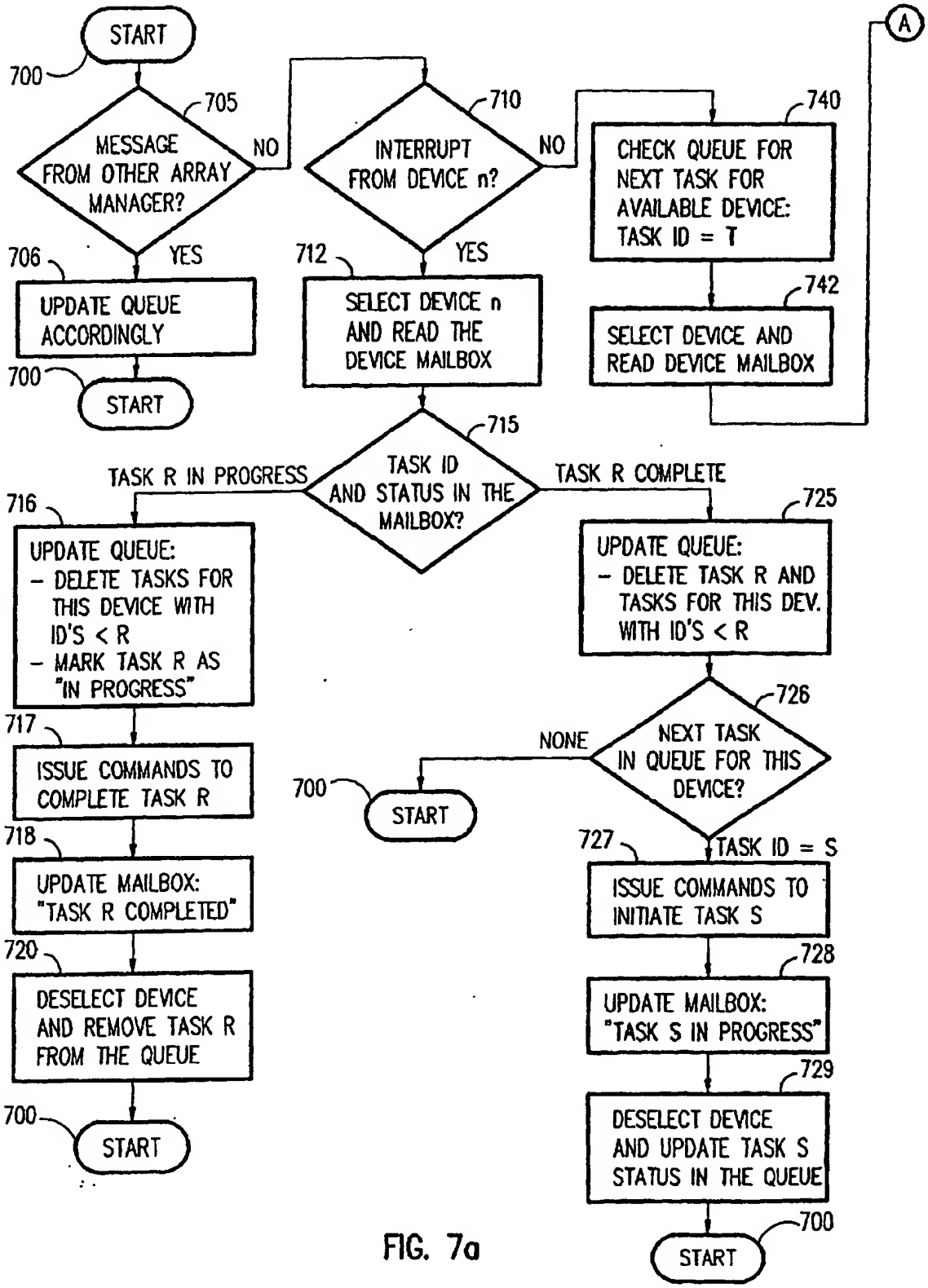

START
700
705
MESSAGE FROM OTHER ARRAY MANAGER?
NO
YES
706
UPDATE QUEUE ACCORDINGLY
700
START
710
INTERRUPT FROM DEVICE n?
NO
YES
712
SELECT DEVICE n AND READ THE DEVICE MAILBOX
715
TASK ID AND STATUS IN THE MAILBOX?
TASK R IN PROGRESS
TASK R COMPLETE
716
UPDATE QUEUE:
- DELETE TASKS FOR THIS DEVICE WITH ID'S < R
- MARK TASK R AS "IN PROGRESS"
717
ISSUE COMMANDS TO COMPLETE TASK R
718
UPDATE MAILBOX: "TASK R COMPLETED"
720
DESELECT DEVICE AND REMOVE TASK R FROM THE QUEUE
700
START
740
CHECK QUEUE FOR NEXT TASK FOR AVAILABLE DEVICE: TASK ID = T
742
SELECT DEVICE AND READ DEVICE MAILBOX
A
725
UPDATE QUEUE:
- DELETE TASK R AND TASKS FOR THIS DEV. WITH ID'S < R
726
NEXT TASK IN QUEUE FOR THIS DEVICE?
NONE
700
START
TASK ID = S
727
ISSUE COMMANDS TO INITIATE TASK S
728
UPDATE MAILBOX: "TASK S IN PROGRESS"
729
DESELECT DEVICE AND UPDATE TASK S STATUS IN THE QUEUE
700
START

FIG. 7a

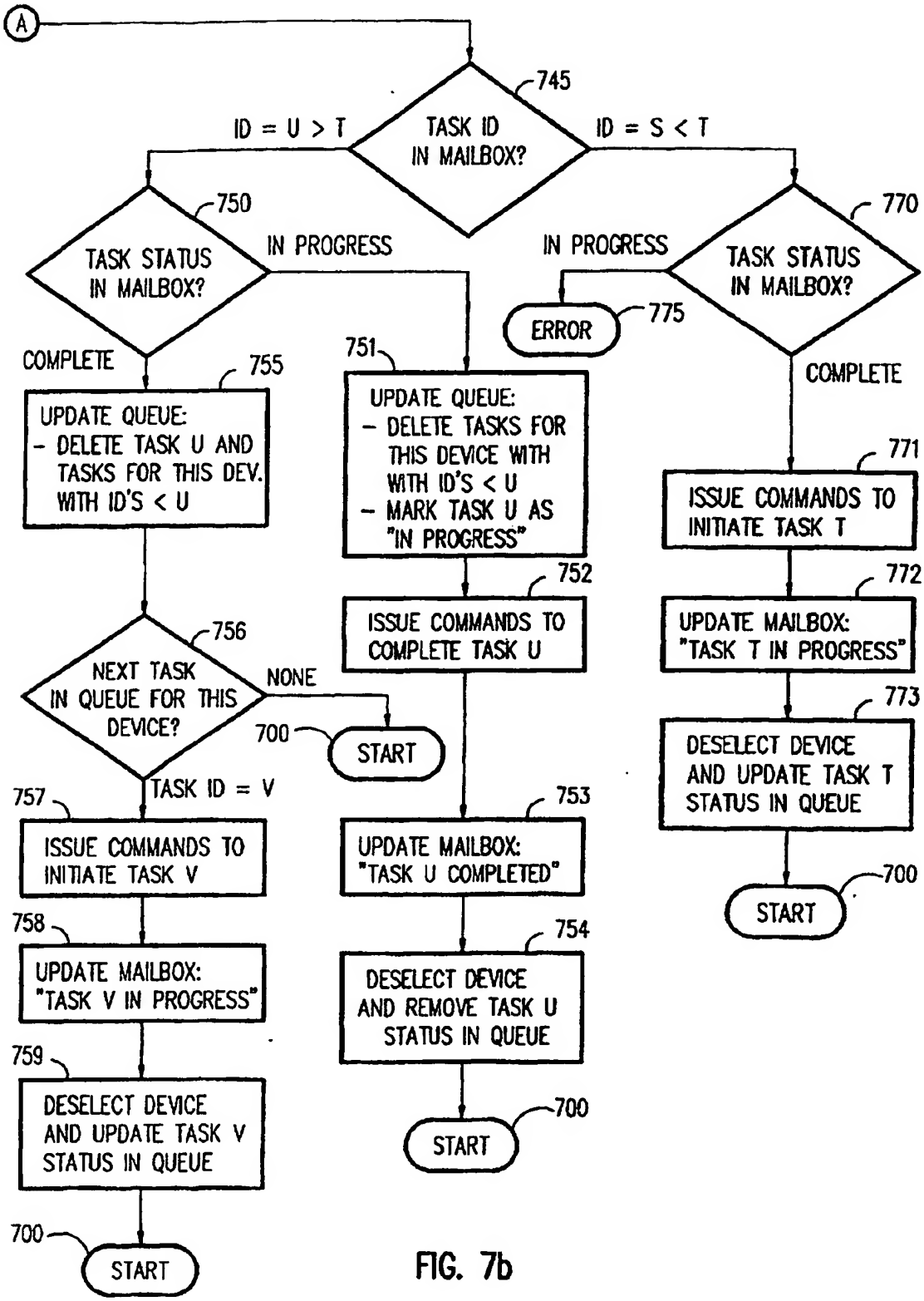

A
745
TASK ID IN MAILBOX?
ID = U > T
ID = S < T
750
TASK STATUS IN MAILBOX?
IN PROGRESS
COMPLETE
755
UPDATE QUEUE:
- DELETE TASK U AND TASKS FOR THIS DEV. WITH ID'S < U
756
NEXT TASK IN QUEUE FOR THIS DEVICE?
NONE
700
START
TASK ID = V
757
ISSUE COMMANDS TO INITIATE TASK V
758
UPDATE MAILBOX: "TASK V IN PROGRESS"
759
DESELECT DEVICE AND UPDATE TASK V STATUS IN QUEUE
700
START
751
UPDATE QUEUE:
- DELETE TASKS FOR THIS DEVICE WITH WITH ID'S < U
- MARK TASK U AS "IN PROGRESS"
752
ISSUE COMMANDS TO COMPLETE TASK U
753
UPDATE MAILBOX: "TASK U COMPLETED"
754
DESELECT DEVICE AND REMOVE TASK U STATUS IN QUEUE
700
START
IN PROGRESS
ERROR
775
770
TASK STATUS IN MAILBOX?
COMPLETE
771
ISSUE COMMANDS TO INITIATE TASK T
772
UPDATE MAILBOX: "TASK T IN PROGRESS"
773
DESELECT DEVICE AND UPDATE TASK T STATUS IN QUEUE
700
START

FIG. 7b

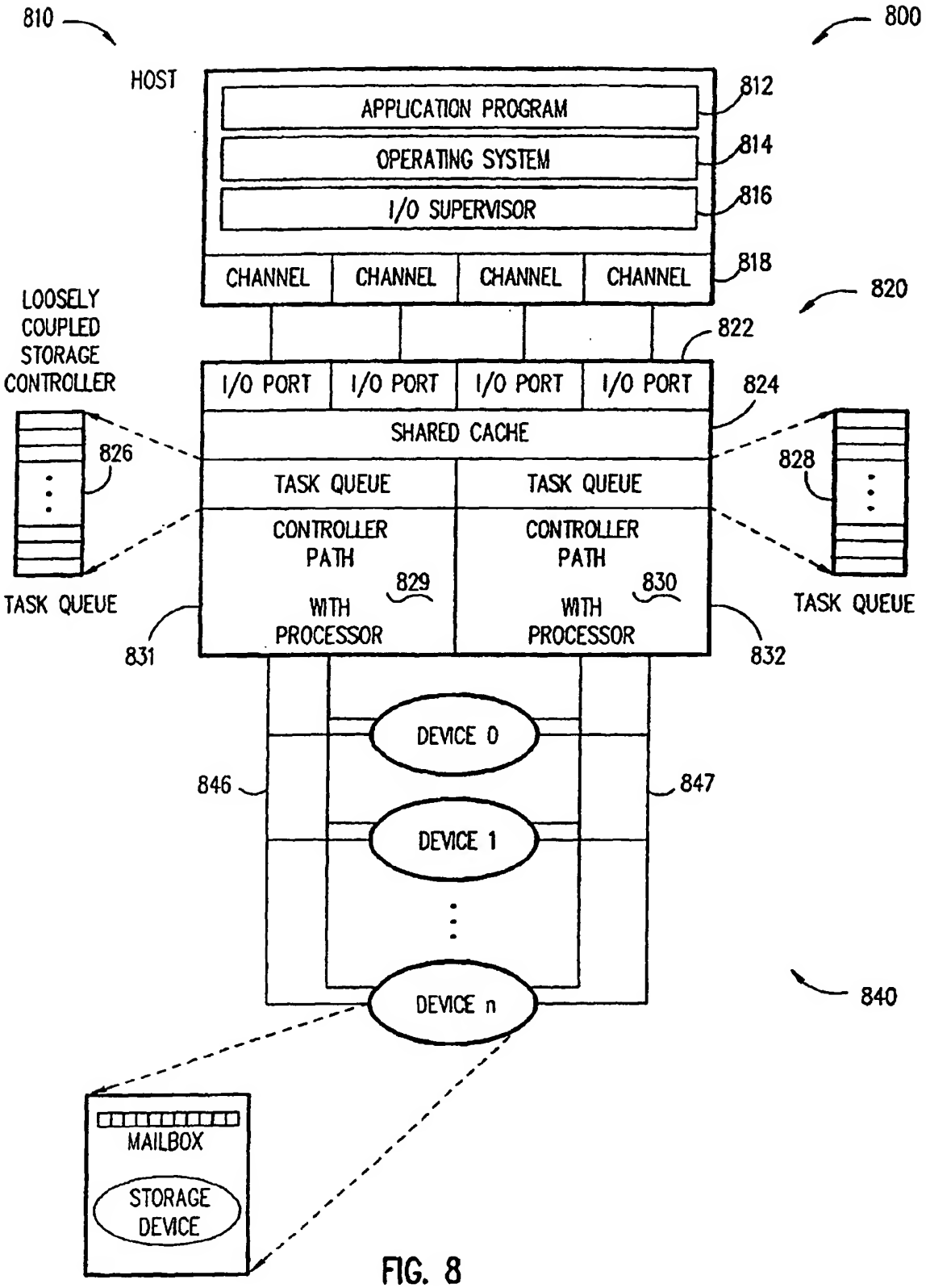

810
800
HOST
APPLICATION PROGRAM
812
OPERATING SYSTEM
814
I/O SUPERVISOR
816
CHANNEL
CHANNEL
CHANNEL
CHANNEL
818
820
LOOSELY COUPLED STORAGE CONTROLLER
822
I/O PORT
I/O PORT
I/O PORT
I/O PORT
824
SHARED CACHE
826
828
TASK QUEUE
TASK QUEUE
CONTROLLER PATH WITH PROCESSOR
829
CONTROLLER PATH WITH PROCESSOR
830
TASK QUEUE
TASK QUEUE
831
832
DEVICE 0
846
847
DEVICE 1
DEVICE n
840
MAILBOX
STORAGE DEVICE

FIG. 8

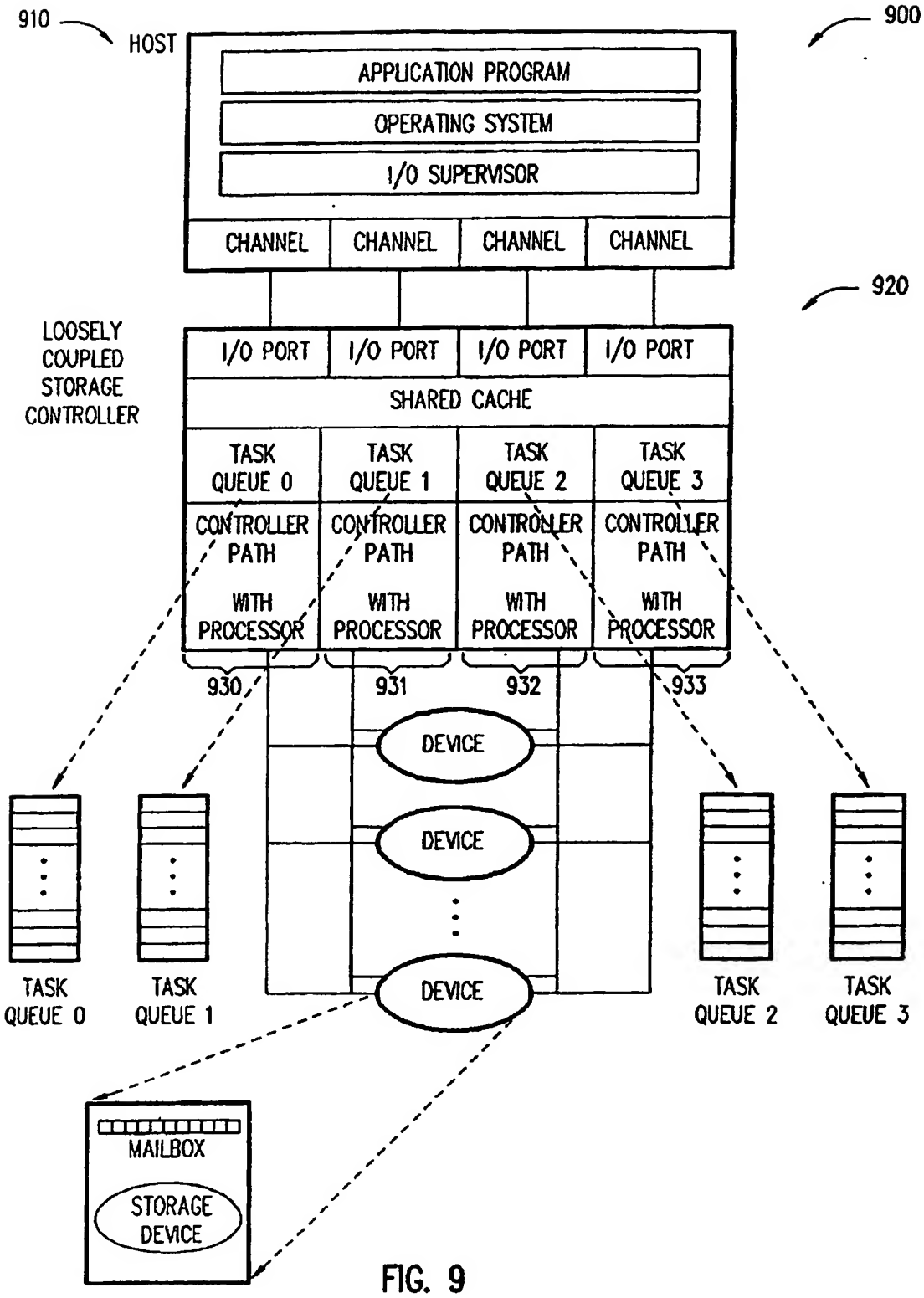

910
HOST
900
APPLICATION PROGRAM
OPERATING SYSTEM
I/O SUPERVISOR
CHANNEL
CHANNEL
CHANNEL
CHANNEL
920
LOOSELY COUPLED STORAGE CONTROLLER
I/O PORT
I/O PORT
I/O PORT
I/O PORT
SHARED CACHE
TASK QUEUE 0
TASK QUEUE 1
TASK QUEUE 2
TASK QUEUE 3
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
CONTROLLER PATH WITH PROCESSOR
930
931
932
933
DEVICE
DEVICE
DEVICE
TASK QUEUE 0
TASK QUEUE 1
TASK QUEUE 2
TASK QUEUE 3
MAILBOX
STORAGE DEVICE

FIG. 9

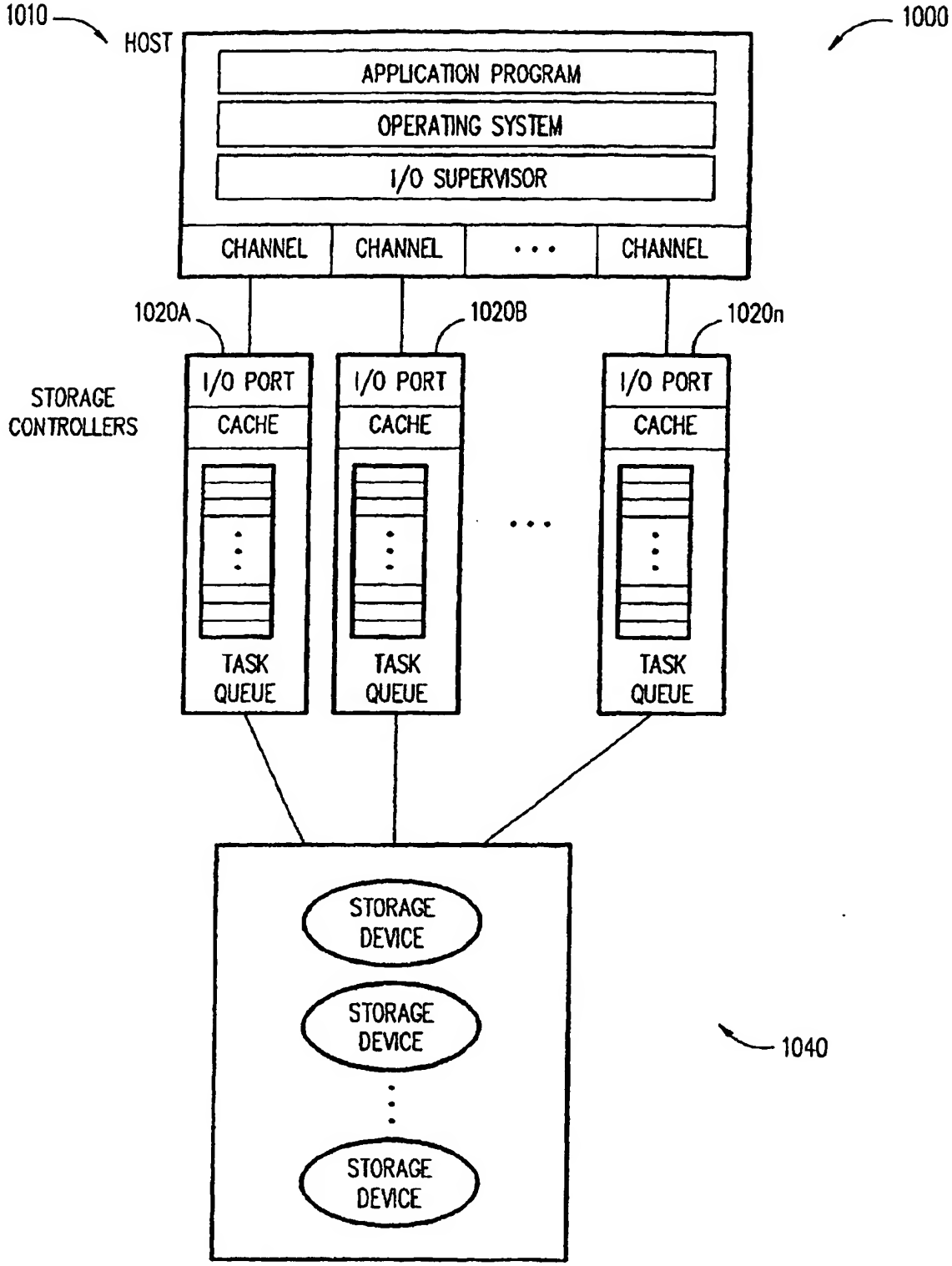

1010
HOST
1000
APPLICATION PROGRAM
OPERATING SYSTEM
I/O SUPERVISOR
CHANNEL
CHANNEL
• • •
CHANNEL
1020A
1020B
1020n
STORAGE CONTROLLERS
I/O PORT
CACHE
TASK QUEUE
I/O PORT
CACHE
TASK QUEUE
• • •
I/O PORT
CACHE
TASK QUEUE
STORAGE DEVICE
STORAGE DEVICE
STORAGE DEVICE
1040

FIG. 10

Microsoft® Windows Server System™

Highly Available Storage:

Multipathing and the Microsoft MPIO Driver Architecture

White Paper

Published: October 2003

Abstract

High reliance on the Internet and e-commerce means that organizations are increasingly requiring that their data be available 24x7, year round. Multipathing, the ability of a system to use more than one read/write path to a storage device, is a high availability solution that provides fault tolerance against single-point-of failure in hardware components. Multipathing can also provide load balancing of I/O traffic, thereby improving system and application performance.

This white paper is of interest to the storage administrator who is ultimately responsible for ensuring that an organization's data remains highly available, and to storage vendors who target multipathing solutions using the Microsoft MPIO (multipath input/output) solution in Windows 2000 Server, Windows Server 2003, and Windows Storage Server 2003. The paper begins with a discussion of the situations and environments in which multipathing is a smart business solution, then describes the critical factors that must be part of an effective multipathing solution, and concludes with a brief overview of the driver-level details of the Microsoft MPIO solution.

The Microsoft MPIO driver development kit is not available as part of the Windows Driver Development Kit; it can be licensed by hardware and software vendors following execution of a Program Agreement. Organizations that seek Windows-based multipathing solutions should check with their vendors to ensure that their storage devices are Microsoft MPIO-compatible.

Contents

Introduction

The most commonly implemented high availability storage solution is to provide disk redundancy through using a redundant array of independent disks (RAID). While RAID-1 (mirroring) provides excellent fault tolerance, it only protects the disks. If there is only a single path from the server to the storage device, however, and one of the components in that path fails, no amount of disk redundancy can keep the data available.

Multipathing solutions, in contrast, are designed to provide failover through the use of redundant physical path components—adapters, cables, and switches—between the server and storage device. In the event that one or more of these components fails, applications can still access their data. Fault tolerance is not the only benefit of multipathing solutions; multipathing software also serves to redistribute the read/write load among multiple paths between the server and storage, thereby helping to remove bottlenecks and to balance work loads.

Previously, the Windows operating system did not have a native Windows architecture to provide failover or load balancing features. Without third party software, if more than one path to a storage device existed, there was no way for the operating system to correctly interpret that multiple storage devices were in fact a single unit. Instead, the operating system "saw" as many devices as there were paths to it. However, even with third party software, multipathing solutions in the Windows environment often failed. Developing multipath solutions that work effectively with the operating system is a complex undertaking. It is even more challenging when an organization needs to support storage devices from multiple vendors.

The release of Microsoft's multipathing drivers (Microsoft MPIO) allows software and hardware vendors to develop solutions that are not only specific to their storage devices, but also work effectively with the Microsoft® Windows® 2000 Server and Microsoft® Windows Server™ 2003 platforms—including Microsoft® Windows® Storage Server 2003—to provide high availability and high performance solutions. Microsoft MPIO (multipath input/output) drivers are written and maintained by Microsoft to ensure close interoperability with the Windows operating system. Moreover, Microsoft MPIO solutions are designed to work properly when installed on the same server as an existing Microsoft MPIO solution from another vendor.

High Availability Solutions

Keeping mission-critical data continuously available has long been critical in the enterprise environment, where the goal for many organizations is continuous availability—the so-called "five nines" standard of server availability—99.999 percent uptime or only a few minutes of unplanned downtime in a year of 24X7 access. Even for enterprises, this level of availability is very difficult and very costly to achieve. High availability, defined as 99.9% uptime (which translates into several hours downtime per year), is a more attainable goal. Both standards require that redundancy be built in at all levels: disk redundancy, backups to separate recovery servers, server clustering, and redundancy of the path components.

Application Availability through Server Clustering

Clustering is the use of multiple computers, interconnects, and storage devices that work together to provide users with high application availability from what looks like a single system. If a computer system goes down, or is temporarily unavailable, end users can still access their data following failover to an active system. Clustering solutions require software that enables transparent failover between systems. Microsoft Cluster Service (MSCS) is one such solution and it is included with Windows Server 2003, Enterprise Edition, and Windows Server 2003, Datacenter Edition.

Storage Redundancy through RAID

High availability solutions based on redundant arrays of independent disks (RAID) have been in use with mainframe equipment for several decades, and have been supported in the Windows operating system since the first release of Windows NT. RAID solutions provide protection through the use of redundant disks, which can be configured for both fault tolerance and improved performance. (For a discussion of how RAID solutions work in Windows Server 2003 and Windows Storage Server 2003, see the white paper "Storage Management using Windows Virtual Disk Service and Volume Shadow Copy Service," issued September 2003.)

Storage Availability through Multipathing

Although both multipathing and clustering result in high availability and improved performance, they are not equivalent concepts. While clustering provides high application availability and can be implemented using a pathing solution, multipathing provides high storage availability and does not require a clustering solution.

Multipathing is redundancy of the storage network components only—the cabling, adapters, and switches—that transfer data across the path between the server and storage. Multipathing solutions manage these redundant connections so that read/write requests can be instantaneously rerouted in case a path fails.

Because consolidated resources are inherently at higher risk, implementation of multipathing solutions is especially important for storage administrators when storage resources are consolidated on network attached storage (NAS) devices or storage area networks (SANs).

How Data Becomes Unavailable

To design highly available server, network, and storage infrastructures, it is necessary to understand the ways in which applications can fail to access storage. Failure to access to data can be ascribed to two sources:

- The root underlying cause—the failure mode—whether software or hardware
- The component where the point of failure occurs

Failure Modes

Failure modes are the ways in which a system (or a system component or process) can potentially fail. Servers can become unavailable for a variety of reasons, ranging from component failures, configuration problems, application and system errors, and excessive network traffic, to just plain human error, such as accidentally unplugging a cable.

Whether or not a failure mode is fatal depends both on the type of failure (some failures can be re-tried and the read/write request will then succeed) and whether or not the hardware of a specific vendor is designed to help with error recovery. In the case of fatal errors where there is no chance that the I/O request can succeed, a multipathing solution ensures that the I/O request can be attempted and completed on a different path.

- **Data corruption**. Often caused by hardware faults or by incorrect software or firmware, incorrect data is read from or written to media, usually without a recorded indication of an error. This constitutes the most serious server failure mode, since it is almost impossible to troubleshoot; these types of errors are usually fatal.
- **File system corruption**. Typically due to a failed I/O that results in a partial update of data, this type of fault leaves the file system in an inconsistent state. With a journaled file system, such as NTFS, this error is usually detected; however, it is possible that the application state cannot be rolled back, and the error can result in loss of data. In some cases, the error can be severe enough to result in loss of the entire volume.
- **Specific hardware errors**. An I/O fails due to a recognized hardware error. Whether or not these errors are fatal depends on the specific hardware error. Since these faults can be tracked, the risk for this type of error can be minimized.
- **I/O Problems**. Read/write requests remain active, yet fail to reach completion. These difficulties can be caused by improperly coded internal retry queues within different components of the driver stack, such as vendor-supplied multipathing drivers, class drivers, mini-ports or port drivers. These errors usually require manual intervention and can result in loss of data. I/O requests can also fail completely. It is not generally possible to determine the exact nature of these failures beyond general device I/O failure. These errors are usually fatal.
- **Soft errors**. These are correctable (non-fatal) errors, such as disk block read problems, which might either be retried or reconstructed automatically. For long-term reliability, it is desirable to keep track of these errors. In many cases, if frequent soft errors are encountered, the application may not be able to keep up with the workload and then performance is lowered to unacceptable levels.

Points of Failure

Each component in the I/O path is a potential point of failure. Such components include the server (host) components, storage path components—such as HBAs, cable and switches—and the storage array or its components. In addition, each component, while it may not completely or permanently fail itself, still has the potential to impact layers upstream in such a way that I/O failure, improper path failover or catastrophic faults may result.

Host Components

Server failure can result from a number of common failure modes, including hardware failure and software failure (I/O timeouts or application induced errors). Servers can be protected against such failures by providing redundancy through clustering.

- **Bus.** Most hardware platforms now have multiple buses. Consequently, redundant hardware (from interface down) should be segregated to different buses wherever possible. While bus failure is not common, it is common for a device on a bus to cause problems for other interfaces (such as improper PCI behavior or bad driver behavior). Except in the case of hardware failure, it is generally difficult to determine exactly why a single bus device is malfunctioning (whether because of I/Os timeouts or failed I/O); however, disabling one or more devices on a bus can allow the server to continue to function. Most bus failures will be seen as a server failure except on the highest end server lines.
- **Interface.** The main goal of multipathing is to allow multiple interfaces to the same storage unit (LUN). Here, interface refers to an I/O card in a bus, such as a Fibre Channel adapter. Although fairly reliable, failures—stemming from hardware problems, I/O problems, timeouts, and data corruption—do occur. For highest availability, servers should use redundant HBAs, and possibly Hot-Plug PCI systems with compatible adapters and drivers, allowing these components to be replaced without bringing down the server.

Storage Path Components

There are numerous components in the storage path that can fail. For this reason, multipathing is a particularly critical solution for ensuring highly available data.

- **GBIC and SFP.** Common physical interface components in Fibre Channel SANs are the GBIC (gigabit interface converter) or the newer SFP (small form factor package, pluggable transceiver), modules that convert electrical signals to be propagated via fibre optics or copper cables. Historically, GBIC modules have had a high failure rate, although SFPs represent a substantial reliability improvement. Failures are generally manifest as either complete loss of signal on a path or very high error rates. Following a module failure, path failover should occur, either automatically or manually once the bad module has been located. Replacement of the GBIC or SPF does not require server reboots because these devices are inherently hot pluggable (unless there is only a single path to the device behind the GBIC/SFP, in which case reboot is required).
- **Cabling.** Cable failures are very common for out-of-cabinet storage. Poor contact on large multi-conductor parallel SCSI cables and dirty optics or overstressed fibers on fiber cables are not unusual. Resulting faults may or may not result in path failure, although almost certainly such faults will result in high error rates which can result in diminished performance (throttling). The most likely failure modes are manifested as hardware problems or and I/O timeouts.
- **Switches.** Fibre Channel SANs can be configured using either hubs or switches. Both switches and hubs can fail because of faulty hardware or network connectivity problems. Enterprises are usually discouraged from using hubs as they have a number of drawbacks, including shared bandwidth, and possible management complexity. Additionally, hubs can create addressing problems if there is a disruption of a Fibre Channel Arbitrated Loop.

  Switches, on the other hand, can provide several advantages: Each port generally has full bandwidth (point-to-point); switches are readily managed (and in fact often have an embedded management server that provides management functions). Moreover, depending on the model and configuration, switches can prevent disruption of Fibre Channel arbitrated loop. Much of this functionality is price related: Low-end switches, for instance, act like hubs—they may not provide high bandwidth performance and they still rely on arbitration. Mid-level switches provide higher bandwidth and include management

features; some include redundant power supplies as well. High-end switches (also called "Director" class) can provide complete fault tolerance with highly redundant components and automatic path failover within the switch, as well as the ability to update the switch firmware without disruption I/O. Note, however, that a switch failover can result in other components, such as drivers, misbehaving. These switches require the same level of intelligent path management as would high-end RAID storage units.

Storage Devices and Subsystems

Storage devices—whether JBODs (just a bunch of disks) or RAID storage arrays—can fail as a result of hardware problems, I/O problems, timeouts, data corruption, or soft errors.

- **RAID subsystems**. For enterprise discussions, high-end RAID subsystems are the norm. These systems are highly redundant, with multiple ports, processors and power supplies. Failure modes of these units merit very careful attention when designing recovery behaviors. Some troublesome areas are controller failover (either due to error or temporary disruption during updating procedures), RAID performance (particularly during disk error recovery) and spare drive behavior (when a spare becomes active, one or more LUNs may suddenly become read-only in the middle of operations).
- **JBOD**. For some applications, JBOD arrays are specified, but they too must be constructed with redundant components such as power and cooling. JBODs will generally not have as complicated a failure mechanism or recovery as do the RAID boxes. Fibre Channel disks have two paths ("dual-ported") which can improve fault tolerance and performance, but multipathing must be used to take advantage of those capabilities.

Multipathing Solutions

Windows multipathing solutions can be designed into the operating system or can be designed by third party vendors. In either case, the multipathing solution consists of one or more *drivers*—software that is designed to manage the hardware components on behalf of the operating system.

Third Party Solutions

One of the greatest complexities in developing effective device drivers is ensuring both that they are compatible with the operating system and that they do not block the functionality of other vendor storage devices or drivers. Third party multipathing drivers are usually provided by storage array manufacturers, although in some cases they are developed by the manufacturers of host bus adapters or by storage software vendors.

The multipathing solutions developed by third party hardware manufacturers have the advantage of expert knowledge of their storage subsystem; read/write requests to the storage device can be optimized, for example. Vendor knowledge, however, does not usually extend to the architecture of the operation system kernel or storage stack. One of the negative consequences of lacking such knowledge is that the drivers they develop can fail to work properly. In the Windows operating system environment, this translates into a number of very serious problems, including operating system crashes, failure to load under all conditions such as "Safe mode" boot, and poor Plug and Play functionality—any of which can result in loss of data or data corruption.

Another serious problem with a number of vendor device drivers is that, depending on how they are designed, interoperability with other vendor storage devices may be prevented. This is a serious problem for storage administrators who are either locked into a single storage vendor, or are faced with serious and often insurmountable configuration problems when attempting to use storage or pathing components from multiple vendors.

Operating System-Based Solutions

There are a number of advantages to an operating system-based (native) multipathing solution. Foremost among them is that a solution that is designed to be integrated with the operating system prevents the problems often associated with third party drivers. Another enormous advantage to the customer is that, through a host-based multipathing solution, it is possible to ensure interoperability of storage devices from multiple vendors.

The drawback to operating system-based multipathing solutions is that intimate knowledge of the specific capabilities of the vendor device is not possible.

A Joint Solution: MPIO and Device-Specific Modules

The Microsoft MPIO solutions are designed to work in conjunction with device specific modules (DSMs) written by vendors; the MPIO driver package does not, by itself, form a complete solution. Microsoft provides a sample device-specific module, or DSM, designed to provide a software interface between the multipath driver package and the hardware device. Vendors must adapt this generic DSM to the specifics of their device or devices.

This joint solution allows vendors to design hardware solutions that are tightly integrated with the Windows operating system, and also enables Microsoft to correctly accommodate the non-generic characteristics of each vendor's storage device (such as whether there are multiple active controllers or the controllers have only standby capability), without having to design the MPIO solution in anticipation of each possible difference. Compatibility with both the operating system and other vendor storage devices is ensured through requiring that

vendors meet a set of standards (the Microsoft Logo program) designed help ensure proper vendor device functionality. The bottom line for the customer is that a multipathing solution based on Microsoft MPIO provides a high performance solution that keeps data highly available.

Making Multipathing Solutions Work

The Windows operating system relies on the Plug and Play (PnP) Manager to dynamically detect and configure hardware (such as adapters or disks), including hardware used for high availability/high performance multipathing solutions. PnP devices are automatically detectable, and can be hot plugged into a PnP bus without the need to reboot the system. In contrast, legacy devices (such as those that plug into serial or parallel ports) must be manually configured and the system may need to be rebooted for the operating system to register their presence correctly. Dynamic discovery reduces both management complexity and the potential for configuration errors, both of which are especially detrimental in large or complex storage configurations. The Microsoft MPIO driver development kit is designed to work seamlessly with PnP architecture.

PnP devices are under the control of PnP drivers that understand the plug and play protocols of the Windows operating system. Control of the device itself rests with lower layer (vendor) drivers.

Device Discovery and Enumeration

Multipathing solutions cannot work effectively unless the redundant adapters connecting to the same storage device can be discovered, enumerated, and configured correctly—that is, without the operating system erroneously interpreting multiple paths as leading to multiple storage devices. Figure 1 shows what happens with and without multipathing software in place.

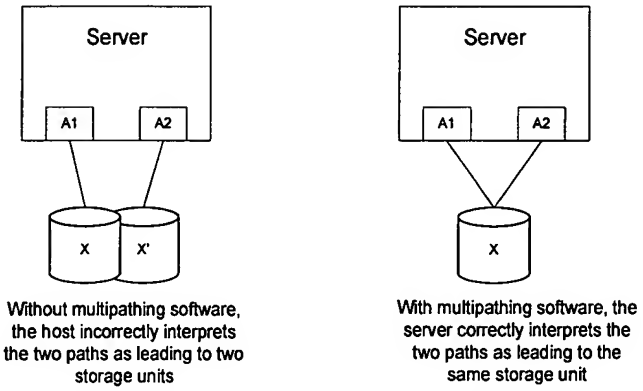


Figure 1. What the operating system "sees" with and without multipathing software

Each device must first be identified as belonging to a specific vendor, and then a determination is made as to whether the device ID represents a unique device or the same underlying physical device accounted for through another path.

Unique Hardware Identifier

For dynamic discovery to work correctly, some form of identifier must be identified and obtainable regardless of the path from the host to the storage device. Each logical unit must have a unique hardware identifier. The MPIO driver package does not use "disk" signatures placed in the data area of a disk for identification purposes, but instead uses standard information obtainable from the hardware, such as serial numbers.

Since not all vendors assign their devices a unique hardware serial number, Microsoft includes in its sample generic DSM source code a means of deriving one, using other SCSI INQUIRY data. Alternatively, vendor-specific mechanisms can be implemented in the DSM.

Dynamic Load Balancing

Load balancing, the redistribution of read/write requests for the purpose of maximizing throughput between server and storage device, is especially important in high workload settings or other settings where consistent service levels are critical. Without multipathing software, a server sending I/O requests down several paths may operate with very heavy workloads on some paths while others are underutilized.

The Microsoft MPIO software supports the ability to transparently balance I/O workload, without administrator intervention. MPIO determines which paths to a device are in an active state and can be used for load balancing. Each vendor's load balancing policy (which may use any of several algorithms, such as round robin, the path with the fewest outstanding commands, or a vendor unique algorithm) is set in the DSM. This policy determines how the I/O requests are actually routed. If the DSM returns a path that is inactive, the failover process is initiated.

Error Handling, Failover and Recovery

The MPIO driver package, in combination with the vendor DSM, supports end-to-end path failover. The process of detecting failed paths and recovering from the failure, like load balancing, is automatic, extremely fast, and completely transparent to the IT organization. The data remains available at all times.

Not all errors result in failover to a new path. Some errors are temporary and can be recovered using a recovery routine in the DSM; if recovery is successful, MPIO is notified and path validity checked to verify that it can be used again to transmit I/O requests.

When a fatal error occurs, the path is invalidated and a new path is selected. To do this correctly, outstanding I/Os must be returned to a higher level in the stack. Once this is done, all devices on that path can be removed and then failed over to the new path.

The Windows Storage Stack and Drivers

The last section of this paper presents an overview of the low level details of the Microsoft MPIO driver architecture. This information is more in-depth than is required by readers who only seek an introduction to the Microsoft MPIO architecture.

For the operating system to correctly perform operations that relate to hardware, such as addition or removal of devices or transferring I/O requests from an application to a storage device, the correct device drivers must be associated with the device. All device-related functionality is initiated by the operating system, but under direct control of subroutines contained within each driver. These processes are considerably complicated when there are multiple paths to a device. Multipathing software prevents data corruption by ensuring correct handling of the driver associated with a single device that is visible to the operating system through multiple paths.

Storage Stack

Storage architecture in Windows consists of a series of layered drivers, as shown in Figure 2. (Note that the application and the disk subsystem are not part of the storage layers.) When a device such as a storage disk is first added in, each layer of the hierarchy is responsible for making the disk functional (such as by adding partitions, volumes and the file system (FS)). The stack layers below the broken line are collectively known as the device stack and deal directly with managing storage devices.

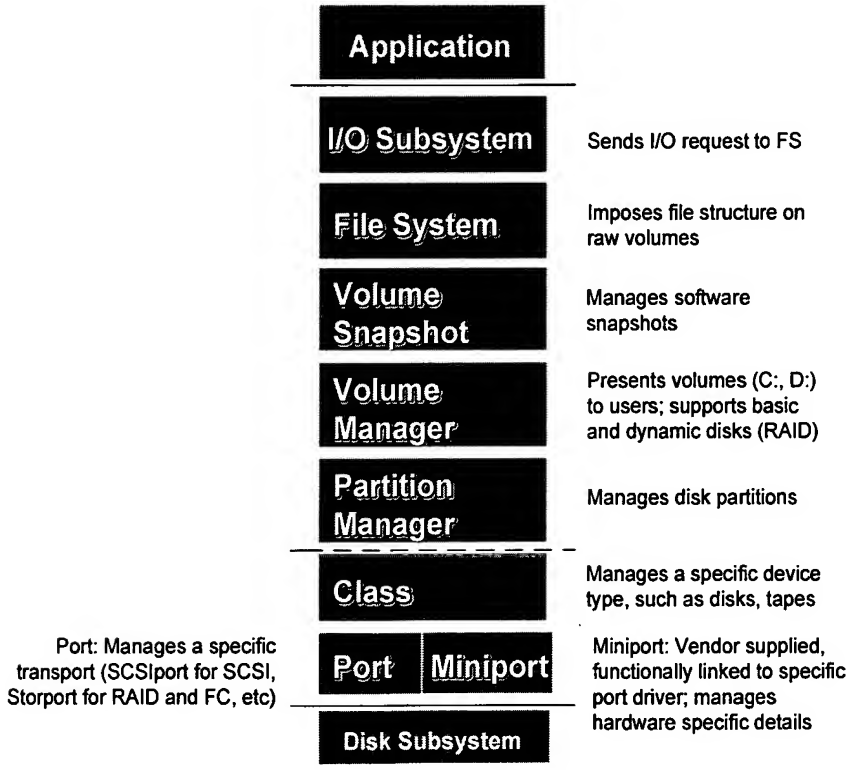


Figure 2. The Windows Storage Stack

Once the disk is ready for use, applications can then send read/write (I/O) requests from the server down to the hardware storage device.

Device Drivers

Device drivers manage specific hardware devices, such as a disks or tapes, on behalf of the operating system.

Port Drivers

Port drivers manage different types of transport, depending on the type of adapter (USB, SCSI or Fibre Channel (FC), for example) in use. One of the most common port drivers in the Windows system is the SCSIport driver. In conjunction with the class driver, the port driver handles PnP and power functionality. Port drivers create a physical device object (PDO), which is used to represent the connection between the device and the bus. Windows Server 2003 introduced a new port driver, Storport, which is better suited to high-performance, high-reliability environments.

Miniport Drivers

Each storage adapter has an associate device driver known as a "miniport." This driver implements only those routines necessary to interface with the storage adapter's hardware. A miniport combines (links) with a port driver to implement a complete layer in the storage stack, as shown in Figure 2.

Class Driver

Class drivers manage a specific device type. Class drivers are responsible for presenting a unified disk interface to the layers above (for example, to control read/write behavior for a disk). The class driver creates the function device object (FDO) used to manage the functionality of the device. Class drivers (like port and miniport drivers) are not a part of the MPIO driver package per se; however, the Plug and Play disk class driver, **disk.sys**, is used as part of the multipathing solution, since the driver controls the disk add/removal process and I/O requests pass through this driver to the MPIO bus driver (see "MPIO Drivers," the following section).

MPIO Drivers

The MPIO driver package consists of three multipath drivers: the port filter driver, the disk-driver replacement, and the bus driver; all of which are implemented in the kernel mode of the operating system. The MPIO driver package works in combination with both the PnP Manager, the disk class driver, the port driver, the miniport driver, and each vendor's device-specific module (DSM) to provide full multipathing functionality.

Multipath Port Filter Driver (mpspfltr)

The multipath port filter driver loads between the port driver and the class driver, and manages information passed up the stack by the port drivers. The filter changes the default ID for disk devices from the generic one of "GenDisk" to the MPIO specific "MPIODisk".

Multipath Disk Driver Replacement (mpdev.sys)

The multipath disk driver replacement is a class driver. Once a device is identified and the DSM of a specific vendor is associated with it, mpdev.sys claims ownership of the disk device object (the PDO) and prevents any other driver from being associated with it. This directly prevents another driver from creating another device stack on the same LUN and mounting a new file system—a process that would destroy the original file system and any data on the LUN. Without this driver, this disk class driver would claim each instance of the disk devices, creating multiple active paths to the device. This driver also notifies the MPIO bus driver of new device arrivals.

Multipath Bus Driver (mpio.sys)

Bus drivers are responsible for managing the connection between the device and the host computer. The multipath bus driver is a "root bus"—the conceptual analog to an actual bus slot into which a device plugs. It acts as the parent bus for the multipath children (disk PDOs). As a root bus, mpio.sys can create new device objects that are not created directly by new hardware being added into the configuration. The MPIO bus driver also communicates with the other multipath drivers, manages the PnP connection and power control between the hardware devices and the host computer, and provides a method for vendors to monitor and manage their storage and associated DSMs.

MPIO DSM

The MPIO driver package includes generic code for vendors to adapt to their specific hardware device so that usage and performance of the device can be improved. Device-specific information is abstracted and exported to the bus driver and to the disk objects under its control.

Each DSM plays a role in a number of critical events, including device-specific initialization, request handling, and error recovery (retrying and failover).

Device Initialization

Each DSM is contacted in turn to determine whether or not it provides support for a specific device. If the DSM does support the device, it then indicates whether the device is a new installation, or the same device previously installed but which is now visible through a new path.

Request Handling

When an application makes an I/O request to a specific device, the DSM makes a determination, based on its internal load balancing algorithms, as to which path the request should be sent. If the I/O request cannot be sent down a path because the path is broken, the DSM will shift to error handling mode.

Error Handling

The DSM determines whether to retry the I/O request, or to treat the error as fatal, making fail-over necessary. In the case of fatal errors, paths are invalidated, and the request is rebuilt and transmitted to a different device path.

DSM Management

Management and monitoring of the DSM can be done through an administrative utility. The preferred interface is through the Windows Management Instrumentation (WMI). The MPIO development kit does not provide management utilities.

Conclusion

The Microsoft MPIO architecture provides hardware and software vendors with a means of creating Microsoft MPIO solutions that work effectively with the Windows Server operating system to provide organizations with reliable I/O path support on Windows 2000 Server, Windows Server 2003, and Windows Storage Server 2003 platforms. The current Microsoft MPIO driver development kit provides support for Fibre Channel and parallel SCSI interfaces. Future releases will provide multipath support for other devices, including tapes and iSCSI connected devices.

Organizations of all sizes can now benefit from increasingly flexible multipathing solutions in Windows environments. Once only affordable by enterprise organizations (who generally purchased end-to-end storage solutions from a single vendor), high availability and high performance multipathing storage solutions are increasingly in demand from midsized organizations whose storage network configurations routinely consist of multi-vendor devices.

By providing a single interface through which vendors each develop their own device-specific multipathing solutions, the Microsoft MPIO architecture not only ensures compatibility within current and future Windows platforms, but also prevents the multi-vendor device incompatibilities that have plagued previous multipathing solutions. These advantages bring multipathing solutions to midsized and small businesses, and offer enterprise customers greater flexibility in the storage solutions they adopt, allowing them to maintain existing enterprise solutions while also adding less expensive storage solutions to support storage management scenarios.

Resources

Information about the Windows Hardware Quality Labs is available at http://www.microsoft.com/whdc/hwtest/

Products designed for use with Windows are listed at http://www.microsoft.com/whdc/hcl/default.mspx

Information about Windows Management Instrumentation (WMI) is available on the following Microsoft Web sites:

- http://www.microsoft.com/whdc/hwdev/driver/WMI/default.mspx
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/wmi_start_page.asp
- http://msdn.microsoft.com/library/default.asp?url=/library/en-us/kmarch/hh/kmarch/wmi_2dyf.asp

Windows Server System is comprehensive, integrated, and interoperable server infrastructure that simplifies the development, deployment, and operation of flexible business solutions.
www.microsoft.com/windowsserversystem